Progress of Concurrent Objects with Dynamic Dependencies

MO ZOU, Shanghai Jiao Tong University, China

Progress of blocking objects is challenging to verify due to the possible dependencies between progress of concurrent threads. Existing program logics for progress verification have limited support for nested and dynamic dependencies of progress, which are often found in real-world objects with fine-grained locking. This paper proposes a new program logic PODD to modularly verify progress and linearizability of concurrent objects. Our key ideas, dynamically layered definite actions, enable modular reasoning by allowing the verifiers to layer the dependency chains between progress of threads on the fly. PODD employs dependency enforcement to ensure the absence of unsound circular dependencies. With PODD, we have, for the first time, modularly verified representative concurrent objects that exhibit nested and dynamic progress dependencies, as well as several concurrent list algorithms with fine-grained locking whose proofs in existing program logics are complex and not even modular.

1 INTRODUCTION

A concurrent object exposes a set of methods for client threads to access the shared data. Besides linearizability [Herlihy and Wing 1990], a safety property, it should also satisfy *progress* properties. This paper focuses on progress of *blocking* objects, where the delay of a thread can prevent other threads from making progress. Many concurrent objects are blocking, including all lock-based algorithms, such as Michael and Scott's two-lock queues [Michael and Scott 1996] and lock-coupling lists [Herlihy and Shavit 2008]. They should satisfy starvation-freedom or deadlock-freedom as progress properties, ensuring that every or some method call can terminate under fair scheduling.

Verifying progress of blocking objects is challenging because progress of one thread may depend on progress of others. Consider a simple object with a global queue lock `L`. Any thread that hopes to access the data needs to first acquire `L`, and all threads trying to acquire `L` form a queue. A thread `t` can acquire `L` and finish its method only when the threads ahead of `t` in the queue can acquire `L` and then release `L` in order. So there are dependencies between progress of `t` and other threads. The main verification challenge is allowing this kind of "good" dependencies while ensuring the absence of circular dependencies that would cause deadlocks.

For objects with multiple locks, the progress dependencies can be *nested*, making modular verification extremely hard. Fig. 1 shows a pattern of nested locking (ignore the code in gray boxes for now), which creates *nested dependencies* of progress. The object has three public methods using two locks `A` and `B`, whose implementations are internal modules of the object. Due to the nested locking in `AB()`, the acquisition of lock `A` of one thread may depend on the release of *the other lock* `B` of other threads. Specifically, consider the client program `A()` ‖ `AB()` ‖ `B()`, executed by the threads t_A, t_{AB} and t_B. It has the following execution: first t_{AB} acquires `A` and t_B acquires `B`, then t_A and t_{AB} have to spin inside their code of `lock A` and `lock B` to wait for the two locks, respectively. Consequently, t_A's acquisition of `A` depends on t_{AB}'s release of `A`, and the latter requires t_{AB} to acquire `B`, which further depends on t_B's release of `B`. In this case, the dependencies of progress are not only between different threads requesting the same lock but also between the acquisition of different locks. Modular verification would hope to verify the acquisitions of different locks *independently*, but it is challenging to achieve due to the nested dependencies of progress.

Some objects further exhibit *dynamic dependencies* of progress. In their executions, multiple method calls may request the locks in different orders, so the dependencies between locks cannot be statically determined. We will use Linux file systems and *transfer lists* (a realistic example designed by us) to illustrate the practical need for dynamic dependencies in Sec. 2.3. Here, as a simple (though contrived) case, we extend the object in Fig. 1 with the code in gray boxes. Now, `BA()` requests the locks of `A` and `B` in the reverse order from `AB()`, which means the dependencies

```
A(){ lock A; unlock A; }

AB(){
  lock L ; lock A; lock B;
  unlock L ; unlock A; unlock B;
}
```

```
B(){ lock B; unlock B; }

BA(){
  lock L; lock B; lock A;
  unlock L; unlock B; unlock A;
}
```

Fig. 1. A simple nested locking object, with nested and dynamic dependencies of progress.

between locks are dynamically changed. Despite the use of lock `L` in `AB()` and `BA()` to prevent deadlocks, the interference on dependencies may still affect other methods where `A` and `B` are involved in dependencies with other locks, e.g., lock `C` in (1.1). The general question is how to ensure the absence of circular dependencies such that a method (e.g., `CA()`) can terminate even when concurrent executions may dynamically change dependencies between locks.

```
CA(){ lock C; lock A; unlock C; unlock A; }
```
(1.1)

Previous efforts [Balzer et al. 2019; Jacobs et al. 2022] for deadlock-freedom verification support some form of dynamic dependencies. Their approaches assume primitive blocking constructs, e.g., lock primitives. However, ad-hoc synchronizations, like the busy-waiting patterns, are common in fine-grained programs. It is unclear how to apply the techniques for general progress reasoning with combinations of locks, ad-hoc synchronizations, and dynamic dependencies between them.

Existing program logics for progress reasoning have limited support for nested and dynamic dependencies. LiLi [Liang and Feng 2016, 2018] introduces the idea of *definite actions* to handle dependencies of progress between threads. But for nested locking objects (Fig. 1), LiLi requires one to always unfold and expose the full dependency chains, thus cannot independently verify the acquisitions of different locks. TaDA Live [D'Osualdo et al. 2021] introduces *layered obligations* to reason modularly about nested dependencies. However, it requires the layers to be statically specified, and it is unclear how to handle the cases with dynamically changed dependencies. We will discuss more related work in Sec. 6.

In this paper, we propose a new program logic PODD [1], which, for the first time, supports modular verification of progress that possibly involves nested and dynamic dependencies. PODD verifies concurrent objects under fair scheduling and unifies the verification of linearizability and starvation-freedom/deadlock-freedom. Our work is based on earlier efforts on concurrency verification and progress verification but makes the following new contributions:

- We propose *dynamically layered definite actions*. The fulfillment of a definite action at a higher layer is allowed to depend on other definite actions at lower layers. This enables concise proofs for nested locking objects by hiding the dependency chains on inner locks when verifying outer locks. The layer of each definite action is defined as a function of states, so it can effectively support dynamic dependencies. To avoid circular dependencies, we enforce the layer relation to be stable between definite actions with actual dependencies.
- We provide a novel *frame* rule over definite actions. It allows us to verify the progress of some code with the definite actions that it relies on and then reuse the verified code in different contexts without redoing the proof. Together with the layering of definite actions, we can modularly reason about individual locks in nested locking objects.
- We prove the soundness of PODD. Despite the dynamically changing layers, we can still find a decreasing metric as evidence for progress. PODD establishes termination-preserving simulations between object implementations and their atomic specifications. It ensures a

[1] PODD is short for Progress of Objects with Dynamic Dependencies.

```
inc:                lock:                          unlock:
  lock;               local i;                       owner := owner + 1;
  x := x + 1;         i := getAndInc(next);
  unlock;             while (i != owner) {};
```

Fig. 2. A counter with a ticket lock.

contextual refinement under fair scheduling, which has been proved equivalent to starvation-freedom/deadlock-freedom and linearizability in earlier work [Liang and Feng 2016].

- We have applied PODD to verify transfer lists as well as the introductory example of Fig. 1. To our knowledge, we are the first to modularly verify the progress of transfer lists, a realistic example that involves dynamic dependencies. We have also verified all the examples that were verified in LiLi. For objects with fine-grained locks, such as linked lists, optimistic lists and lazy lists, our proofs are more concise and modular than those in LiLi.

Outline. We first develop our key ideas in Sec. 2. Then we give the basic technical settings in Sec. 3 and present our logic together with the soundness theorem in Sec. 4. We discuss the examples we have verified in Sec. 5. We discuss related work and conclude in Sec. 6.

2 DYNAMICALLY LAYERED DEFINITE ACTIONS

The core idea of our PODD is the dynamically layered definite actions. Below we first review the (non-layered) definite actions that had been used in LiLi for verifying progress of blocking objects (Sec. 2.1). Then we introduce layers to handle nested dependencies of progress (Sec. 2.2) and explain the practical need for dynamic layers (Sec. 2.3).

2.1 Background: Proving Progress of Blocking Objects with Definite Actions

For blocking objects, there are two common progress properties: starvation-freedom (SF) and deadlock-freedom (DF) [Herlihy and Shavit 2011]. Both SF and DF assume fairness of execution, which roughly says that every thread gets eventually executed. SF requires that *every* thread can finish its method call in fair executions, and DF requires that there always *exists some* thread that can finish its method call in a fair execution.

Example: a single-lock counter. As a simple example of blocking objects, Fig. 2 shows a counter with one public method `inc()`. It uses a ticket lock to protect the shared data `x`. The ticket lock ensures the first-come-first-served property using two shared variables: `next` records the next available ticket number, and `owner` records the ticket number of the thread that currently owns the lock. Initially, `next` and `owner` are both 0. To acquire the lock, a thread first atomically reads and increments `next` to get its ticket number `i`. Then it waits until `owner` equals `i`. Later the thread can release the lock by atomically incrementing `owner`, letting the next waiting thread hold the lock.

This object ensures SF for two reasons. First, with fairness, any thread that owns the lock can *definitely* release the lock and finish its method call. Second, any blocked thread (i.e., a thread that spins inside the **while**-loop of `lock`) waits for only a *finite* sequence of lock releases. As a result, every thread can eventually finish its method call in a fair execution, so the object satisfies SF.

Definite actions. To formulate the above intuition, Liang and Feng [2016] propose definite actions in LiLi, which will eventually happen, regardless of the environmental interference.

Specifically, a definite action $\mathcal{D}$ is in the form of $P \leadsto Q$, where P and Q are both state assertions. "Definite" requires that *if* P holds, (1) Q should eventually be established, and (2) P should be preserved (by both the environment and the current thread) until Q holds. The second condition ensures that the only way to make P false is to fulfill the definite action. We say that $\mathcal{D}$ is *enabled*

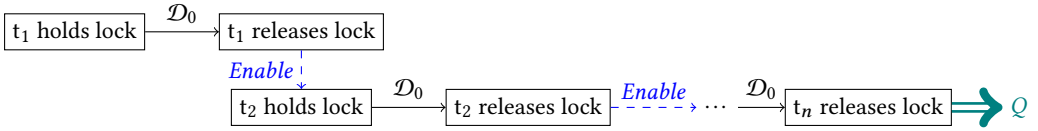

Fig. 3. Definite progress for the counter. $\boxed{\mathtt{t}_k \text{ holds lock}} \stackrel{\text{def}}{=} (\mathtt{owner} = \mathtt{t}_k.\mathtt{i})$ and $\boxed{\mathtt{t}_k \text{ releases lock}} \stackrel{\text{def}}{=} (\mathtt{owner} = \mathtt{t}_k.\mathtt{i} + 1)$ for every $\mathtt{t}_k$. $Q \stackrel{\text{def}}{=} (\mathtt{owner} = \mathtt{t.i})$ for the current thread t. $\mathcal{D}_0$ is defined in the text.

(written as Enabled($\mathcal{D}$)) if P holds. For instance, the definite action $\mathcal{D}_0$ for Fig. 2 is defined as $(\mathtt{owner} = \mathtt{t.i}) \rightsquigarrow (\mathtt{owner} = \mathtt{t.i} + 1)$, where t.i denotes the local variable i of thread t. Here, the thread ID serves as an implicit parameter for all the assertions, including state assertions and definite actions, which is useful for specifying them.

Then, following the intuition of Fig. 2, to prove SF of an object, there are two proof principles on definite actions: (1) ensure *strict definiteness* of definite actions and (2) prove *definite progress* from definite actions. They talk about the termination of definite actions and programs, respectively.

Ensure strict definiteness of definite actions. "Strict" requires that the environment *cannot* block an enabled definite action. In other words, once enabled, an definite action's fulfillment *cannot* rely on the fulfillment of other definite actions. The strictness clears the worries about circular dependencies between definite actions. With the "definite" requirement, one can soundly rely on definite actions to create progress.

For example, the definite action $\mathcal{D}_0$ for Fig. 2 is strictly definite because after thread t acquires the lock, the following codes naturally terminate, and the thread can release the lock by itself.

Prove definite progress from definite actions. The definite progress condition is used for proving termination of loops. Consider the loop of lock in Fig. 2. It must terminate if $(\mathtt{owner} = \mathtt{t.i})$ holds for the current thread t. Otherwise, t is blocked, and the definite progress condition should be proven to hold: it waits for a finite chain of definite actions, as shown in Fig. 3. Here, Q specifies the condition when t can progress on its own, i.e., it is not blocked. Every time a thread $\mathtt{t}_k$ fulfills its definite action, the next thread $\mathtt{t}_{k+1}$'s definite action becomes enabled, and the chain becomes shorter. So eventually, Q will hold, and t will be unblocked.

More formally, the definite progress condition is in the form of $(\mathcal{D}' \xrightarrow{f} Q)$ (a simplified version). Here $\mathcal{D}'$ is the set of definite actions that the current thread waits for and should be a subset to the set of specified definite actions $\mathcal{D}$, written as $\mathcal{D}' \leqslant \mathcal{D}$. f specifies a well-founded metric (i.e., a metric that cannot decrease indefinitely), counting the number of $\mathcal{D}'$-s remained to be fulfilled by the environment threads $\mathtt{t}_k$ before the current thread t is unblocked. The metric decreases whenever one of these $\mathcal{D}'$-s is finished and never increases under any transitions. If the metric has decreased to the minimum, Q must hold. This way, we ensure that t is not permanently blocked. For the loop of lock in Fig. 2, $\mathcal{D}'$ is specified as $\mathcal{D}_0$. The metric f can be defined as $(\mathtt{t.i} - \mathtt{owner})$. It decreases whenever an environment thread $\mathtt{t}_k$ increases owner. When f becomes 0, the loop terminates.

Remark. So far, we have focused on SF. The mechanism for deadlock-freedom follows LiLi and is orthogonal to our main contributions. We will explain it and full logic details in Sec. 4.

2.2 Supporting Nested Dependencies with Layers

Our development so far is general enough to verify various blocking objects, but the reasoning is often *not* modular for objects with *nested dependencies* of progress. Below we first give real-world examples that involve nested dependencies of progress. Then we explain the problem in modular reasoning and address the problem by extending definite actions with layers.

2.2.1 Real-world examples. Nested dependencies of progress show up in nested locking objects. Besides the example in Fig. 1, nested locking is commonly found in many real-world concurrent objects that use fine-grained locks. A thread often has to acquire multiple locks to accomplish an operation. E.g., a process in Linux can acquire up to 12 locks in file system operations or an average of 4 locks in memory management [Park et al. 2021]. Another example is two-phase locking in databases and transaction processing, where a lock protects each data item. The expanding phase of the two-phase locking protocol only acquires locks and often performs nested locking.

2.2.2 Problems with strictly definite actions. Can we use the approach in Sec. 2.1 to verify nested locking objects, such as Fig. 1? Let's assume the locks are ticket locks and ignore the code in gray boxes throughout Sec. 2.2. Modular verification would hope to verify the acquisitions of multiple locks independently. In the proof for `lock A`, we rely on definite actions specifying the release of `A`, but we need *not* be aware of the existence of other locks. A natural proposal for Fig. 1 would let each definite action (e.g., $\mathcal{D}_A/\mathcal{D}_B$) specify the release of each lock (e.g., `A`/`B`) separately.

However, this ideal proposal of definite actions does not meet the *strict definiteness* requirement in Sec. 2.1, which requires that the fulfillment of an enabled definite action should be by itself. For instance, if a thread t has acquired `A` in the `AB()` method of Fig. 1, it may not be able to fulfill $\mathcal{D}_A$ because it may get blocked when trying to acquire `B`. In this case, the fulfillment of $\mathcal{D}_A$ depends on the release of `B` by other threads, which violates strict definiteness. So it seems that we cannot use $\mathcal{D}_A$ and $\mathcal{D}_B$ to verify Fig. 1. Nevertheless, it *is* possible to verify Fig. 1 using the approach in Sec. 2.1. We can unfold the dependency chain and treat every action in the flattened chain as a definite action. Then the new definite actions are naturally strictly definite.

For instance, Fig. 4a shows such a flattened chain. Here the current thread t is blocked in the loop of `lock A`. The environment threads t_A and t_{AB} execute `A()` and `AB()`, respectively, taking smaller ticket numbers than t during `lock A`. So t waits for t_A and t_{AB} to release `A`. Once t_A releases `A` (see $\mathcal{D}_1$), t_{AB} owns `A`. Now t_{AB}'s definite action (see $\mathcal{D}_2$) gets enabled, which says that t_{AB} must request `B` (the corresponding code is the `getAndInc` in Fig. 2, for lock `B`). It is the next action after `lock A` in `AB()`, so it is strictly definite. After finishing it, t_{AB} enters the loop of `lock B`. If thread t_B has acquired `B` in `B()`, t_{AB} would be blocked. Once t_B releases `B` (see $\mathcal{D}_3$), t_{AB} acquires `B` and its next definite action is enabled, which is the release of `A` after acquiring both `A` and `B` (see $\mathcal{D}_4$). In the end, the current thread t is unblocked, and Q holds.

These strictly definite actions are very fine-grained. With them, when proving the definite progress condition for the loop inside `lock A`, we would have to go through the flattened chain of definite actions, including the ones on the release of `B`. The proofs would be unnecessarily complex.

2.2.3 Layers to the rescue. Definite actions could have dependencies, e.g., for `AB()` in Fig. 1, the release of `A` ($\mathcal{D}_A$) could depend on the release of `B` ($\mathcal{D}_B$). But we need to ensure $\mathcal{D}_B$ does not depend backward on $\mathcal{D}_A$ to avoid unsound circular dependencies. For this, our approach is to layer the definite actions and only allow higher-layer definite actions to depend on lower-layer ones.

Layered definite actions. We introduce a layer function $\mathfrak{L}$, mapping each basic definite action $\mathcal{B}$ to a layer. $\mathcal{B}$ is an element from the set of definite actions $\mathcal{D}$, and we may also use $\mathcal{D}$ to refer to a basic definite action. Layers are elements from a *totally ordered* set, such as *Nat*. For now, we assume that an object has a finite number of definite actions, e.g., $\mathcal{D}_0, \ldots, \mathcal{D}_n$, and $\mathfrak{L}$ gives each a static layer. With layered definite actions, the proof principles become: (1) ensure *dependent definiteness* of definite actions and (2) prove definite progress from definite actions with *lower layers*.

Ensure dependent definiteness of definite actions. "Dependent" requires that the fulfillment of $\mathcal{D}$ is allowed to depend on some other $\mathcal{D}'$, as long as $\mathfrak{L}(\mathcal{D}) > \mathfrak{L}(\mathcal{D}')$, i.e., the layer of $\mathcal{D}$ is *strictly higher* than the layer of $\mathcal{D}'$. Consequently, the ordering between layers determines dependencies allowed

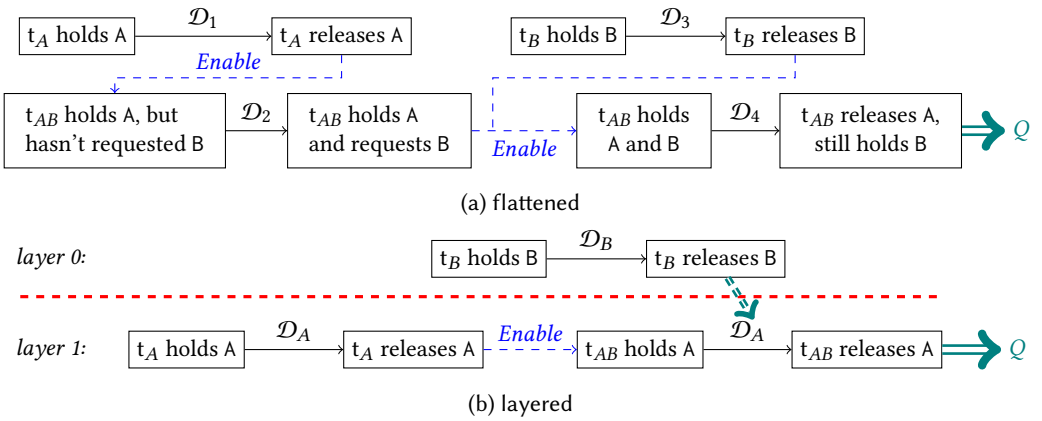

Fig. 4. Definite progress for the nested locking object in Fig. 1 (without the code in gray boxes). Here $Q \stackrel{\text{def}}{=} (\texttt{owner} = \texttt{t.i})$ for the current thread t.

between definite actions. Since layers are totally ordered, we avoid circular dependencies. Notice that for a definite action $\mathcal{D}_0$ at the *lowest* layer, dependent definiteness implies strict definiteness, i.e., $\mathcal{D}_0$ should definitely happen on its own once enabled. For Fig. 1, we can now use definite actions $\mathcal{D}_A$ and $\mathcal{D}_B$. We let the layer function $\mathfrak{L}$ assign layer-1 to $\mathcal{D}_A$ and layer-0 to $\mathcal{D}_B$. So, $\mathcal{D}_A$ can depend on $\mathcal{D}_B$, but not vice versa. $\mathcal{D}_B$ must be strictly definite.

Definite progress from lower-layer definite actions. When blocked in loops, we prove the definite progress condition, which roughly says with the help of definite actions $\mathcal{D}'$, the thread will reach the unblocked state specified by Q. Previously, an enabled definite action could *not* be blocked, so $\mathcal{D}'$ can be chosen freely from $\mathcal{D}$. Now we allow enabled definite actions to be blocked. To obey dependent definiteness, $\mathcal{D}'$ should satisfy the new requirement $\mathcal{D}' \leqslant_{J,\mathfrak{L}} \mathcal{D}$ as follows.

$$\begin{array}{l}\mathcal{D}' \leqslant_{J,\mathfrak{L}} \mathcal{D} \ \text{ iff } \ \mathcal{D}' \leqslant \mathcal{D} \wedge \mathsf{selD}_{J,\mathfrak{L}}(\mathcal{D}', \mathcal{D}) \\ \mathsf{selD}_{J,\mathfrak{L}}(\mathcal{D}', \mathcal{D}) \ \text{ iff } \ \forall \mathsf{t}, \mathfrak{S}.\ (\mathfrak{S} \models J_{\mathsf{t}}) \implies \\ \qquad \forall \mathcal{B} \leqslant \mathcal{D}, \mathcal{B}' \leqslant \mathcal{D}'.\ (\mathfrak{S} \models \mathsf{Enabled}(\mathcal{B}_{\mathsf{t}})) \wedge (\exists \mathsf{t}' \neq \mathsf{t}.\ \mathfrak{S} \models \mathsf{Enabled}(\mathcal{B}'_{\mathsf{t}'})) \implies \mathfrak{L}(\mathcal{B}) > \mathfrak{L}(\mathcal{B}')\end{array} \tag{2.1}$$

It requires $\mathcal{D}'$ to still be a subset of $\mathcal{D}$ and satisfy the selection condition selD. selD says at any state $\mathfrak{S}$ satisfying the assertion J (the state consists of store and heap), for any basic definite actions $\mathcal{B}$ and $\mathcal{B}'$ from $\mathcal{D}$ and $\mathcal{D}'$, $\mathcal{B}$ must have a *strictly higher* layer than $\mathcal{B}'$, if $\mathcal{B}$ and $\mathcal{B}'$ are enabled for the current thread t and some other thread t′, respectively.

Consider `AB()` of Fig. 1. For the loop of `lock A`, J specifies that current thread t requests for A, and $\mathcal{D}'$ is $\mathcal{D}_A$. selD holds vacuously becuase either t has not acquired A and has *no* enabled definite actions ($\mathfrak{S} \models \mathsf{Enabled}(\mathcal{B}_{\mathsf{t}})$ does not hold), or t has acquired A so $\mathcal{D}'$ is *not* enbled on other threads ($\exists \mathsf{t}' \neq \mathsf{t}.\ \mathfrak{S} \models \mathsf{Enabled}(\mathcal{B}'_{\mathsf{t}'})$ does not hold). For the loop of `lock B`, J specifies that t owns A and requests for B. $\mathcal{D}'$ is $\mathcal{D}_B$. If another thread owns B, selD holds because $\mathcal{D}_A$'s layer is *strictly higher* than $\mathcal{D}_B$. Otherwise, selD holds vacuously because $\mathcal{D}_B$ is *not* enabled on other threads.

Dependent definiteness allows modular reasoning for nested locking objects. Consider the earilier discussion for Fig. 4a. The new layered dependency chain is in Fig. 4b, where the blocked t waits for $\mathcal{D}_A$ only. Once t_A releases A, the next thread t_{AB}'s $\mathcal{D}_A$ gets enabled. Once t_{AB} fulfills its $\mathcal{D}_A$, t is unblocked, and Q holds. In this reasoning, we *assume* (rather than prove) the successful fulfillment of $\mathcal{D}_A$ of t_{AB} and t_A. The fulfillment of $\mathcal{D}_A$ (after enabled) is proved inside the verification of `A()` and `AB()`. For `AB()`, the fulfillment of $\mathcal{D}_A$ requires the thread to terminate the loop of `lock B`, which relies on $\mathcal{D}_B$ of other threads (see the green dashed arrow across the layers in Fig. 4b).

Soundness. Definite progress ensures that the current thread can reach Q and then progress by itself, *as long as* the selected definite actions $\mathcal{D}'$ can be finished after enabled. So the key question is, can every enabled $\mathcal{D}$ (at any layer) be eventually finished? Yes, the proof is done by induction over $\mathcal{D}$'s layer. For now, assume that (1) an unblocked thread can terminate and (2) enabled definite actions are fulfilled before the thread terminates (formalization presented in Sec. 4).

- **Base case:** $\mathcal{D}$ is at the lowest layer and cannot be blocked. It is fulfilled due to (1) and (2).
- **Inductive step:** suppose $\mathcal{D}$ is at layer n. The induction hypothesis says any layer-m ($m < n$) definite action are fulfilled. Whenever $\mathcal{D}$ is blocked in loops, it will reach the unblocked state due to definite progress and the induction hypothesis. So it is fulfilled according to (1) and (2).

2.3 Supporting Dynamic Dependencies with Dynamic Layers

Our development so far assigns static layers to definite actions, which unfortunately cannot support *dynamic dependencies* of progress. Below we first give examples that involve dynamic dependencies of progress, which motivate our final solution of dynamically layered definite actions.

2.3.1 Real-world examples. For objects with nested locking, the intuitive way to avoid deadlocks is to arrange the same nested locking order globally. But in practice, one often *cannot* do so. Consider Linux file systems. They provide a `move` method (i.e., `rename` syscall), which roughly moves a node from one directory to another and needs to acquire the two directories' locks. For the case that the two directories are different and not ancestors to each other, the code [2022] first acquires a big lock `mutex` to forbid multiple cross-directory `move`s, and then it can acquire the two directories in any order, e.g., the order that arguments are passed. For instance, assume there exist two directories whose paths are `/foo` and `/bar`. A `move(/foo,/bar)` performs `lock mutex;lock foo;lock bar` while `move(/bar,/foo)` acquires `foo` and `bar` in the reverse order. One *should not* modify the code to enforce the same locking order (or *cannot* do so without introducing deadlocks) because it is the natural result of file system logic. Clearly, the dependencies between locks are dynamically changed in Linux file systems (confirmed by Linux documentation [2022]).

Below we show a more approachable object called transfer lists, inspired by file systems. Fig. 5 shows the code for transfer lists. It maintains an array `tl` of lists. Assume there is *no* duplicate data in each list. The internal function `locate(l, e)` (Fig. 5a) finds the node with data `e` in list `l` and returns the node and its previous node. Assume there are two sentinel nodes at the two ends of the list with special value `GUARD` that does not equal any passed argument `e`. `locate` is implemented using lock coupling, i.e., acquiring the next lock and then releasing the lock at hand. Each list `l` (with index `id`) supports public methods `add(id, e)` and `rmv(id, e)` for adding and removing nodes, respectively (code omitted here), which invoke the `locate(l, e)` for list traversal.

The object has a public method `move` to transfer a node from the source list (`src`) to the target list (`dst`) if the node's data does not exist there. `move` picks the first non-sentinel node `y` in `tl[src]` and get its data `e` (lines 3–11). If `e` is not in the target list `tl[dst]` (lines 12–19), `move` then transfers this node to the target list (lines 20–29). Atomicity of the transfer is ensured (including the node removal and insertion) by holding all relevant locks, so it introduces nested locking between the nodes in two lists. Moreover, `move` may acquire locks in different orders . If client threads invoke `move(1,2)` and `move(2,1)` concurrently, we will be at risk of deadlocks. Similar to file systems, `move` uses a lock `mutex` (line 2) to avoid deadlocks. Nevertheless, the interference on dependencies is not contained within `move`s but exposed to other methods (i.e., `add` and `rmv`) that can execute concurrently. The general question for progress verification is how to ensure the absence of circular dependencies despite the dynamically changing dependencies.

```
struct Node {
  int lock;
  int data;
  struct Node *next;
}
struct List {
  struct Node *Head;
}

initialize(){
  Head := cons(0, GUARD, null) ;
  Head.next := cons(0, GUARD, null);
  mutex := 0;
}
```

```
locate(l, e):
    local p, c, u;
 1  p := l.Head;
 2  lock(p);
 3  c := p.next;
 4  u := c.data;
 5  while(u != e && u != GUARD){
 6    lock(c);
 7    unlock(p);
 8    p := c;
 9    c := p.next;
10    u := c.data;
11  }
12  return (p,c);
```

(a) locate in a lock coupling list

```
struct List tl[N];
int mutex;
move(src, dst)
    local x, y, z, e, m, n, u;
    // check src and dst are in range
    // src does not equal dst
 1  ...
 2  lock(mutex);
 3  x := tl[src].head;
 4  lock(x);
 5  y := x.next;
 6  e := y.data;
 7  if (e = GUARD) {
      // if tl[src] is empty
 8    unlock(mutex);
 9    unlock(x);
10    return false;
11  }
12  (m, n) := locate(tl[dst], e);
13  u := n.data;
14  if (u = e) {
      // if e already in tl[dst]
15    unlock(mutex);
16    unlock(x);
17    unlock(m);
18    return false;
19  }
    // remove y in tl[src]
20  lock(y);
21  z := y.next;
22  x.next := z;
    // insert y in tl[dst]
23  y.next := m.next;
24  m.next := y;
25  unlock(mutex);
26  unlock(x);
27  unlock(y);
28  unlock(m);
29  return true;
```

(b) move in transfer lists, where `locate` uses the implementation in (a)

Fig. 5. Pseudocode for transfer lists.

Problems with static layers. Static layers force the dependencies between definite actions to be fixed, which cannot apply to the above examples. E.g., for the full code of Fig. 1 (we target the full code throughout Sec. 2.3), since `AB()` and `BA()` request `A` and `B` in reverse orders, we are unable to assign static layers to $\mathcal{D}_A$ and $\mathcal{D}_B$ so that the dependent definiteness requirement is met.

2.3.2 Dynamic layers. Since dependencies are dynamically changed, the layers should also be dynamically changed. For Fig. 1, a feasible layering strategy may initially assign layer-0 to $\mathcal{D}_A$ and $\mathcal{D}_B$ and layer-2 to $\mathcal{D}_L$. According to the execution, the layers of $\mathcal{D}_A$ and $\mathcal{D}_B$ are changed.

- At the program point after `lock L` in `AB()`, $\mathcal{D}_A$'s layer is increased to layer-1 temporarily.
- At the program point after `lock B` in `AB()`, $\mathcal{D}_A$'s layer is reset to layer-0.
- $\mathcal{D}_B$'s layer is changed similarly in `BA()`.

The ability to assign layers dynamically can effectively handle dynamic dependencies between `A` and `B` and obey dependent definiteness.

Dynamically layered definite actions. We revise the layer function $\mathfrak{L}$ to be a partial mapping from a pair of *state* and definite action to a layer. We can encode the control flow in the state by introducing auxiliary variables and commands. Note $\mathfrak{L}$ is a *partial* function, allowing a definite action's layer to be undefined at some states. This is useful for objects like linked lists where one may dynamically remove a node (and its lock). We write $\mathsf{layDef}(\mathfrak{L}, \mathfrak{S}, \mathcal{D})$ to say that all the definite actions in $\mathcal{D}$ have layers at the state $\mathfrak{S}$. With dynamically layered definite actions, the proof principles then become: (1) ensure dependent definiteness and *dependency enforcement* of definite actions, and (2) prove definite progress from definite actions with *stably* lower layers.

Dependency enforcement. Arbitrary layer changes would allow unsound circular reasoning. For this, dependent definiteness should be immune to layer changes (called *dependency enforcement*). Specifically, recall that dependent definiteness says if the fulfillment of an enabled $\mathcal{D}_1$ depends on the fulfillment of another enabled $\mathcal{D}_2$, $\mathcal{D}_1$'s layer should be *strictly higher* than $\mathcal{D}_2$. Then, dependency enforcement says that although the layers of $\mathcal{D}_1$ and $\mathcal{D}_2$ may dynamically change, their relative layer ordering should be preserved until $\mathcal{D}_2$ is finished.

Definite progress from definite actions with stably lower layers. The definition of selD now considers the state for layer comparison, which naturally does dependency enforcement.

$$\begin{array}{l}\mathcal{D}' \leqslant_{J,\mathfrak{L}} \mathcal{D} \;\text{ iff }\; \mathcal{D}' \leqslant \mathcal{D} \wedge \mathsf{selD}_{J,\mathfrak{L}}(\mathcal{D}', \mathcal{D}) \\ \mathsf{selD}_{J,\mathfrak{L}}(\mathcal{D}', \mathcal{D}) \;\text{ iff }\; \forall \mathsf{t}, \mathfrak{S}.\ (\mathfrak{S} \models J_{\mathsf{t}}) \implies \boxed{\mathsf{layDef}(\mathfrak{L}, \mathfrak{S}, \mathcal{D}') \wedge} \\ \qquad (\forall \mathcal{B} \leqslant \mathcal{D}, \mathcal{B}' \leqslant \mathcal{D}'.\ (\mathfrak{S} \models \mathsf{Enabled}(\mathcal{B}_{\mathsf{t}})) \wedge (\exists \mathsf{t}' \neq \mathsf{t}.\ \mathfrak{S} \models \mathsf{Enabled}(\mathcal{B}'_{\mathsf{t}'})) \implies \\ \qquad \boxed{\mathfrak{L}(\mathfrak{S}, \mathcal{B}) > \mathfrak{L}(\mathfrak{S}, \mathcal{B}')}\ )\end{array} \tag{2.2}$$

The parts in gray boxes are the only differences from (2.1). It asks the layers for definite actions in $\mathcal{D}'$ to be defined and allows the layers of definite actions to change with state, while the relative layer relation between $\mathcal{B}$ and $\mathcal{B}'$ is stable at every state satisfying J.

We will illustrate the definition by extending Fig. 1 with `CA()` in (1.1). Suppose we set the layer of $\mathcal{D}_C$ to layer-2, and the layer of $\mathcal{D}_A$ may dynamically change as explained at the start of Sec. 2.3.2. When proving the definite progress of `lock A` in `CA()`, J specifies that the current thread has acquired `C` and requests for `A`. $\mathcal{D}'$ is $\mathcal{D}_A$. $\mathcal{D}_C$ is enabled and depends on $\mathcal{D}_A$ if `A` is acquired by another thread. Although $\mathcal{D}_A$'s layer may be changed from state to state, we should enforce $\mathcal{D}_C$'s dependency on $\mathcal{D}_A$ ($\mathcal{D}_C$'s layer being *stably higher* than $\mathcal{D}_A$) until $\mathcal{D}_A$ is no longer enabled on other threads, which is what selD requires.

Intuition of soundness. As in Sec. 2.2, the key to logic soundness is to prove that every enabled $\mathcal{D}$ can be eventually finished. In the formal proof, we construct a well-founded metric to ensure that a thread t fulfills its definite action in a finite number of steps. The metric shrinks in several cases.

(1) If t is not blocked and makes a step, the metric decreases.
(2) If t is blocked, we find "the dependency set" consisting of all the threads that t depends on directly or indirectly. If the dependency set shrinks or whenever an unblocked thread in the dependency set makes a step, the metric decreases.
(3) If the dependency set grows, the metric decreases.

We show that the metric keeps decreasing in a fair execution.

- Either the metric decreases according to (1), or t is blocked and waits for threads in the dependency set to execute.
- If the dependency set shrinks, it means some thread in the dependency set has executed and created progress for t. The metric decreases according to (2).

$$
\begin{array}{llll}
(\mathit{Expr}) & E, \mathbb{E} ::= x \mid n \mid E + E \mid \ldots & (\mathit{MName}) & f \in \mathit{String} \\
(\mathit{BExp}) & B, \mathbb{B} ::= \mathsf{true} \mid \mathsf{false} \mid E = E \mid !B \mid \ldots & & \\
(\mathit{Instr}) & c, \mathbb{c} ::= x := E \mid x := [E] \mid [E] := E \mid x := \mathbf{cons}(E, \ldots, E) \mid \mathbf{dispose}(E) \mid \ldots & & \\
(\mathit{Stmt}) & C, \mathbb{C} ::= \mathbf{skip} \mid c \mid x := f(E) \mid \mathbf{return}\ E \mid \langle C \rangle \mid C; C \mid \mathbf{if}\ (B)\ C\ \mathbf{else}\ C \mid \mathbf{while}\ (B)\{C\} & & \\
(\mathit{Prog}) & W, \mathbb{W} ::= \mathbf{skip} \mid \mathbf{let}\ \Pi\ \mathbf{in}\ C \,\|\, \ldots \,\|\, C & (\mathit{ODecl}) & \Pi, \Gamma ::= \{f_1 \rightsquigarrow (x_1, C_1), \ldots, f_n \rightsquigarrow (x_n, C_n)\} \\
(\mathit{Store}) & s, \mathbb{s} \in \mathit{Var} \rightharpoonup \mathit{Int} & (\mathit{Heap}) & h, \mathbb{h} \in \mathit{Nat} \rightharpoonup \mathit{Int} \\
(\mathit{Mem}) & \sigma, \Sigma ::= (s, h) & (\mathit{RelMem}) & \mathfrak{S} ::= (\sigma, \Sigma)
\end{array}
$$

Fig. 6. Syntax of the programming language and states.

- Since layers are totaly ordered, in the dependency set, there exists some thread that has the lowest layer. If the thread t' that has the lowest layer in the dependency set is not blocked, the metric can decrease when t' executes according to (2).
- If t' is also blocked, it can only be blocked by some thread t'' whose layer is *stably lower*. So the dependency set has to be extended with t'', and the metric decreases according to (3). Since the number of threads is bounded [2], the dependency set cannot keep growing.

Finally, the metric must decrease to its minimum value. At that time t's definite action is fulfilled. The full proofs can be found in Appendix.

3 THE LANGUAGE

Fig. 6 presents the syntax of the language. A program W consists of parallel threads, which are the clients of the object Π. The object is declared as a mapping from method names to a pair of an argument and body. The statement C is standard and includes $x := [E]$ and $[E] := E'$ for read and write of the heap at $[E]$, respectively, and $x := \mathbf{cons}(E, \ldots, E)$ and $\mathbf{dispose}(E)$ for allocation and deallocation of the heap, respectively. $\langle C \rangle$ represents an atomic block, where C executes atomically. We use one language for both the low-level implementation and the high-level specification, as shown by the two sets of symbols (e.g., W and $\mathbb{W}$ for low-level and high-level programs, respectively). We make the following simplifications. Assume there is only one object. Each object method takes only one argument and always ends with a **return** command. There are no nested method calls. Each method of the object specification Γ contains an atomic operation $\langle C \rangle$ and **return** command. Assume the specification is safe to execute and never blocks. As an example of the specification, consider the high-level methods that atomically add and remove elements on an abstract set. The corresponding low-level methods may be implemented on a linked list.

Fig. 6 also gives the memory σ, which consists of a store s that maps variable names to values and a heap h that maps addresses to values. Relational memory $\mathfrak{S}$ includes both low-level and high-level memory (σ and Σ). We omit the operational semantics (and the related state for modeling execution) since they are not used in the following development. Full details can be found in Appendix.

4 PROGRAM LOGIC

We extend the program logic LiLi [2016] to modularly verify concurrent objects with nested and dynamic dependencies of progress. LiLi unifies the verification of linearizability and starvation-freedom/deadlock-freedom for concurrent objects. Linearizability is verified by relating each object method to the corresponding atomic abstract operation. LiLi proposes definite actions to handle blocking and ensure starvation-freedom. Deadlock-free objects also allow delay, which is caused by the occurrence of certain actions (e.g., acquiring the lock needed by the current thread) that

[2]Note that the number of threads is a parameter in the logic soundness proof only and can be arbitrary. Users of the logic do not need to be aware of it.

$$
\begin{array}{rrcl}
(\mathit{RelAssn}) & P, Q, J & ::= & B \mid \mathsf{emp} \mid E \mapsto E \mid E \Mapsto E \mid P * Q \mid P \wedge Q \mid P \vee Q \mid \ldots \\
(\mathit{FullAssn}) & p, q & ::= & P \mid \mathsf{arem}(C) \mid \Diamond(E) \mid \blacklozenge(E_k, \ldots, E_1) \mid p * q \mid p \wedge q \mid p \vee q \mid \ldots \\
(\mathit{RelAct}) & R, G & ::= & P \ltimes_k Q \mid [P] \mid \lfloor G \rfloor_0 \mid G \wedge G \mid G \vee G \mid \ldots \\
(\mathit{LayFeat}) & \mathcal{F} & ::= & \ldots \qquad (\mathit{BaseDAct}) \quad \mathcal{B} ::= (P \rightsquigarrow Q, \mathcal{F}) \\
(\mathit{DAct}) & \mathcal{D} & ::= & \mathcal{B} \mid \forall x. \mathcal{D} \mid \mathcal{D} \wedge \mathcal{D} \\
(\mathit{Layer}) & \mathtt{l} & \in & \ldots \qquad (\mathit{LayFun}) \quad \mathfrak{L} \in \mathit{RelMem} \rightharpoonup \mathit{BaseDAct} \rightharpoonup \mathit{Layer}
\end{array}
$$

Fig. 7. Syntax of the assertion language.

make the current thread execute more steps. LiLi uses the token transfer idea to allow delay but ensure whole-system progress. Specifically, each delaying action must consume a token so that a thread cannot be delayed infinitely often without the progress of other threads. The tokens are stratified to handle multiple levels of delaying actions in objects with optimistic synchronization.

However, definite actions in LiLi cannot modularly verify nested dependencies of progress, as discussed in Sec. 2. In PODD, we make the following key extensions to achieve modularity.

- A layer function is specified, which takes the state to express the dynamic layer of a definite action. The definite actions also incorporate information for expressing the layers.
- New principles are enforced: an enabled higher-layer definite action can depend on an enabled definite action with a stably lower layer until the lower-layer one is fulfilled.
- PODD offers a new liveness frame rule over the definite actions and layer function. Specification of a program module only needs to talk about the needed definite actions and corresponding layer mappings in the layer function. Therefore, the progress proof of the module can be reused.

The top-level judgment of PODD is in the form of $\mathfrak{L}, \mathcal{D}, R, G \vdash \{P\}\Pi : \Gamma$ (the OBJ rule for this judgment will be explained later). The object Π is proved to refine the object specification Γ. An object invariant P specifies the consistency relation between the concrete and the abstract data representation. Rely and guarantee conditions (R and G) [Jones 1983] describe the transitions made by the environment threads and the current thread, respectively, and are also lifted to the binary setting. The definite actions $\mathcal{D}$ is a special form of state transitions for progress reasoning, and the layer function $\mathfrak{L}$ maps definite actions to layers dynamically according to the states. Due to space limit, below we will omit the parts of the logic similar to LiLi, and only show the major extensions. The full details of the logic are given in Appendix.

4.1 Assertions

Relational and full assertions. Fig. 7 and Fig. 8 give the syntax and semantics of assertions. Because the logic establishes the refinement between concrete and abstract code, it has relational assertions specifying the relational memory $\mathfrak{S}$. $E \mapsto E$ and $E \Mapsto E$, respectively specify heap cells at the concrete and abstract level. Separating conjunction $*$, as in separation logic, specifies the disjoint union of two parts, which satisfy P and Q, respectively, except now lifted to relational memory. Besides relational memory, the full assertions also describe the abstract code C that remains to be refined (as specified by $\mathsf{arem}(C)$) and the number (u, w) of two kinds of tokens.

Tokens. PODD uses $\Diamond$-tokens to prevent infinite loops and $\blacklozenge$-tokens to prevent infinite delaying actions. A $\Diamond$-token is consumed for each round of a **while**-loop if the thread is not blocked, and each delaying action consumes a $\blacklozenge$-token. Further, the $\blacklozenge$-token is stratified since there are multi-levels of delaying actions. A vector u is used to represent the number of $\blacklozenge$-token at each level, and the consumption of a higher-level token can increase lower-level tokens and $\Diamond$-tokens. The number (u, w) of $\blacklozenge$-token and $\Diamond$-token is specified by $\blacklozenge(E_k, \ldots, E_1)$ and $\Diamond(E)$, respectively.

$$((s,h),(\mathbb{s},\mathbb{h})) \models E_1 \mapsto E_2 \quad \text{iff} \quad h = \{[\![E_1]\!]_{s \uplus \mathbb{s}} \rightsquigarrow [\![E_2]\!]_{s \uplus \mathbb{s}}\}$$
$$((s,h),(\mathbb{s},\mathbb{h})) \models E_1 \Mapsto E_2 \quad \text{iff} \quad \mathbb{h} = \{[\![E_1]\!]_{s \uplus \mathbb{s}} \rightsquigarrow [\![E_2]\!]_{s \uplus \mathbb{s}}\}$$
$$\mathfrak{S} \models P * Q \quad \text{iff} \quad \exists \mathfrak{S}_1, \mathfrak{S}_2.\ (\mathfrak{S} = \mathfrak{S}_1 \uplus \mathfrak{S}_2) \wedge (\mathfrak{S}_1 \models P) \wedge (\mathfrak{S}_2 \models Q)$$
$$(\sigma, \Sigma) \uplus (\sigma', \Sigma') \stackrel{\text{def}}{=} (\sigma \uplus \sigma', \Sigma \uplus \Sigma') \quad \text{where } (s,h) \uplus (s',h') \stackrel{\text{def}}{=} (s \uplus s', h \uplus h')$$

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

$$(\mathfrak{S},(u,w),C) \models P \quad \text{iff} \quad \mathfrak{S} \models P$$
$$(\mathfrak{S},(u,w),C) \models \mathsf{arem}(C') \quad \text{iff} \quad C = C'$$
$$(\mathfrak{S},(u,w),C) \models \Diamond(E) \quad \text{iff} \quad \exists n.\ ([\![E]\!]_{\mathfrak{S}.s} = n) \wedge (n \le w)$$
$$(\mathfrak{S},(u,w),C) \models \blacklozenge(E_k,\ldots,E_1) \quad \text{iff} \quad ([\![E_k]\!]_{\mathfrak{S}.s},\ldots,[\![E_1]\!]_{\mathfrak{S}.s}) \le u$$
$$C \uplus C' \stackrel{\text{def}}{=} \begin{cases} C' & \text{if } C = \textbf{skip} \\ C & \text{if } C' = \textbf{skip} \end{cases} \qquad (\mathfrak{S},(u,w),C) \uplus (\mathfrak{S}',(u',w'),C') \stackrel{\text{def}}{=} (\mathfrak{S} \uplus \mathfrak{S}', (u{+}u', w{+}w'), C \uplus C')$$

(b) Semantics of full assertions p and q.

$$(\mathfrak{S},\mathfrak{S}') \models P \ltimes_k Q \quad \text{iff} \quad (\mathfrak{S} \models P) \wedge (\mathfrak{S}' \models Q)$$
$$(\mathfrak{S},\mathfrak{S}') \models [P] \quad \text{iff} \quad (\mathfrak{S}' = \mathfrak{S}) \wedge (\mathfrak{S} \models P)$$
$$\mathsf{Emp} \stackrel{\text{def}}{=} \mathsf{emp} \ltimes \mathsf{emp} \qquad \mathsf{True} \stackrel{\text{def}}{=} \mathsf{true} \ltimes \mathsf{true} \qquad \mathsf{Id} \stackrel{\text{def}}{=} [\mathsf{true}]$$
$$\mathcal{L}((\mathfrak{S},\mathfrak{S}'), P \ltimes_k Q) \stackrel{\text{def}}{=} \begin{cases} k & \text{if } (\mathfrak{S},\mathfrak{S}') \models P \ltimes_k Q \\ \mathsf{maxL} & \text{otherwise} \end{cases}$$
$$(\mathfrak{S},\mathfrak{S}') \models \lfloor R \rfloor_0 \ \text{ iff } \ \mathcal{L}((\mathfrak{S},\mathfrak{S}'),R) = 0$$
$$(\mathfrak{S},\mathfrak{S}',k) \models R \ \text{ iff } \ \mathcal{L}((\mathfrak{S},\mathfrak{S}'),R) = k \text{ and } k < \mathsf{maxL}$$

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

Fig. 8. Semantics of assertions.

Multi-level rely/guarantee conditions. The rely/guarantee conditions R and G are over relational states transitions and a natural number k, i.e., $(\mathfrak{S},\mathfrak{S}',k)$. The natural number k represents the level $\mathcal{L}$ of actions. We require $0 \le k < \mathsf{maxL}$, where maxL is a predefined upper bound of levels. $P \ltimes_k Q$ is called a level-k transition. It is also called a delaying action when $k > 0$ and a non-delaying action when $k = 0$. The subscript k is omitted when $k = 0$. $[P]$ represents identity transitions with P holding on the states. $\lfloor G \rfloor_0$ can be satisfied only by level-0 transitions in G. Here the definition of level $\mathcal{L}$ for composite rely/guarantee conditions is omitted and can be found in Appendix.

Layered definite actions. Fig. 7 defines layered definite actions, whose semantics is given in Fig. 9. The base form of definite action $\mathcal{B}$ is a pair of $P \rightsquigarrow Q$ and the *feature* $\mathcal{F}$ of the action for deciding the layer. $P \rightsquigarrow Q$ specifies the transitions where the final states satisfy Q *if* the initial states satisfy P. To decide a definite action's layer, the user should find the uniqueness of the action that distinguishes from others. Therefore, we introduce the *layer feature* $\mathcal{F}$ for each definite action to help the user specify the layer. E.g., there are three definite actions for three locks with the same form $\mathcal{B}(\mathsf{x})$ for the object in Fig. 1. $\mathcal{B}(\mathsf{x})$ describes the release of the lock x. The lock that the action operates on gives enough information to decide the layer, therefore, *LayFeat* is instantiated as *Var*, representing the variable of the lock, and x is picked as the layer feature $\mathcal{F}$ of $\mathcal{B}(\mathsf{x})$. We can enumerate the definite action set with $\mathcal{D}_1 \wedge \mathcal{D}_2$ and $\forall x.\ \mathcal{D}$. E.g., the definite actions for Fig. 1 are $\mathcal{B}(L) \wedge \mathcal{B}(A) \wedge \mathcal{B}(B)$.

The layer function $\mathfrak{L}$ is a partial function that takes the relational memory and base definite action, returning a layer if defined. Any type with a total order defined would suffice for expressing the layer, e.g., *Nat*. Taking relational memory as an argument allows defining the layer dynamically. In addition, the layer function should decide the layer solely based on the layer feature of the base definite action. This requirement will be formalized when we discuss the logic rules.

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

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

$$
\begin{array}{rcl}
\mathsf{Enabled}((P \rightsquigarrow Q, \mathcal{F})) & \stackrel{\text{def}}{=} & P \\
\mathsf{Enabled}(\forall x.\, \mathcal{D}) & \stackrel{\text{def}}{=} & \exists x.\, \mathsf{Enabled}(\mathcal{D}) \\
\mathsf{Enabled}(\mathcal{D}_1 \wedge \mathcal{D}_2) & \stackrel{\text{def}}{=} & \mathsf{Enabled}(\mathcal{D}_1) \vee \mathsf{Enabled}(\mathcal{D}_2) \\
\mathsf{layDef}(\mathfrak{L}, \mathcal{S}, \mathcal{B}) & \stackrel{\text{def}}{=} & \exists 1.\, \mathfrak{L}(\mathcal{S}, \mathcal{B}) = 1 \\
\mathsf{layDef}(\mathfrak{L}, \mathcal{S}, \forall x.\, \mathcal{D}) & \stackrel{\text{def}}{=} & \forall n.\, \mathsf{layDef}(\mathfrak{L}, \mathcal{S}\{x \rightsquigarrow n\}, \mathcal{D}) \\
\mathsf{layDef}(\mathfrak{L}, \mathcal{S}, \mathcal{D}_1 \wedge \mathcal{D}_2) & \stackrel{\text{def}}{=} & \mathsf{layDef}(\mathfrak{L}, \mathcal{S}, \mathcal{D}_1) \wedge \mathsf{layDef}(\mathfrak{L}, \mathcal{S}, \mathcal{D}_2) \\
\langle \mathcal{D} \rangle & \stackrel{\text{def}}{=} & \mathcal{D} \wedge (\mathsf{Enabled}(\mathcal{D}) \ltimes \mathsf{true}) \\
[\mathcal{D}] & \stackrel{\text{def}}{=} & \mathsf{Enabled}(\mathcal{D}) \rightsquigarrow \mathsf{Enabled}(\mathcal{D})
\end{array}
$$

$$
\mathcal{D}' \leqslant \mathcal{D} \;\text{ iff }\; \forall \mathcal{B}', \mathsf{genB}(\mathcal{B}', \mathcal{D}') \implies \exists \mathcal{B}, \mathsf{genB}(\mathcal{B}, \mathcal{D}) \wedge \mathcal{B} \equiv \mathcal{B}'
$$

$$
(P \rightsquigarrow Q, \mathcal{F}) \equiv (P' \rightsquigarrow Q', \mathcal{F}') \text{ iff } P \Leftrightarrow P' \wedge Q \Leftrightarrow Q \wedge \mathcal{F} = \mathcal{F}'
$$

$$
\begin{array}{rcl}
\mathsf{genB}(\mathcal{B}, \mathcal{B}') & \stackrel{\text{def}}{=} & \mathcal{B} = \mathcal{B}' \\
\mathsf{genB}(\mathcal{B}, \forall x.\, \mathcal{D}) & \stackrel{\text{def}}{=} & \exists x.\, \mathsf{genB}(\mathcal{B}, \mathcal{D}) \\
\mathsf{genB}(\mathcal{B}, \mathcal{D}_1 \wedge \mathcal{D}_2) & \stackrel{\text{def}}{=} & \mathsf{genB}(\mathcal{B}, \mathcal{D}_1) \vee \mathsf{genB}(\mathcal{B}, \mathcal{D}_2)
\end{array}
$$

(b) Useful syntactic sugars.

Fig. 9. Semantics of layered definite actions.

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

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

$$
\frac{\begin{array}{c}\mathfrak{L}', \mathcal{D}', R, G \vdash \{p\}C\{q\} \qquad \mathcal{D} = \mathcal{D}' \wedge \mathcal{D}'' \qquad \mathfrak{L}' \leqslant \mathfrak{L} \\ p \Rightarrow J \qquad \mathsf{Sta}(J, R \vee G) \qquad \mathsf{layGt}(\mathfrak{L}, \mathcal{D}'', J, \mathcal{D}')\end{array}}{\mathfrak{L}, \mathcal{D}, R, G \vdash \{p\}C\{q\}}\ (\textsc{live-frm})
$$

Fig. 10. Inference rules.

Most syntactic sugars for definite actions are the same as those in LiLi. The notation $\mathcal{D}' \leqslant \mathcal{D}$, which picks a subset $\mathcal{D}'$ from $\mathcal{D}$, is redefined. $\mathsf{genB}(\mathcal{B}, \mathcal{D})$ means the base definite action $\mathcal{B}$ is in (generated from) the definite action set $\mathcal{D}$. The equivalence $\mathcal{B}' \equiv \mathcal{B}$ means their pre-/post-assertions imply each other, and their layer features are equal. $\mathcal{D}' \leqslant \mathcal{D}$ says that for each $\mathcal{B}'$ generated from $\mathcal{D}'$, there is a corresponding $\mathcal{B}$ generated from $\mathcal{D}$, and $\mathcal{B}$ is equivalent to $\mathcal{B}'$.

As in LiLi, all assertions are implicitly parametrized over thread IDs.

4.2 Logic Rules

The OBJ rule. Figure 10 gives the key logic rules. The OBJ rule establishes the termination-preserving refinement between Π and Γ by asking each method C to be simulated by the corresponding abstract operation C'. In the precondition, we assign the number of $\blacklozenge$-tokens through $\blacklozenge(E_k, \ldots, E_1)$. The rule further requires one thread's guarantee to imply other threads' rely, a

$$\mathsf{wffLay}(\mathfrak{L}) \text{ iff } \forall \mathcal{B}_1, \mathcal{B}_2, \mathfrak{S}, \mathtt{l}.\, \mathcal{B}_1.\mathcal{F} = \mathcal{B}_2.\mathcal{F} \implies (\mathfrak{L}(\mathfrak{S}, \mathcal{B}_1) = \mathtt{l} \iff \mathfrak{L}(\mathfrak{S}, \mathcal{B}_2) = \mathtt{l})$$

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

$$\mathsf{layGt}(\mathfrak{L}, \mathcal{D}, J, \mathcal{D}') \text{ iff } \forall \mathfrak{S}.\, \mathfrak{S} \models J \implies (\forall \mathcal{B} \leqslant \mathcal{D}.\, \mathfrak{S} \models \mathsf{Enabled}(\mathcal{B}) \implies (\forall \mathcal{B}' \leqslant \mathcal{D}'.\, \mathfrak{L}(\mathfrak{S}, \mathcal{B}) > \mathfrak{L}(\mathfrak{S}, \mathcal{B}')))$$

$$\mathfrak{L}' \leqslant \mathfrak{L} \text{ iff } \mathsf{subMap}(\mathfrak{L}', \mathfrak{L}) \wedge \mathsf{orderPresv}(\mathfrak{L}', \mathfrak{L})$$

$$\mathsf{subMap}(\mathfrak{L}', \mathfrak{L}) \text{ iff } \forall \mathfrak{S}, \mathcal{B}.\, \mathsf{layDef}(\mathfrak{L}', \mathfrak{S}, \mathcal{B}) \implies \mathsf{layDef}(\mathfrak{L}, \mathfrak{S}, \mathcal{B})$$

$$\mathsf{orderPresv}(\mathfrak{L}', \mathfrak{L}) \text{ iff } \forall \mathfrak{S}, \mathcal{B}_1, \mathcal{B}_2.\, (\mathfrak{L}'(\mathfrak{S}, \mathcal{B}_1) > \mathfrak{L}'(\mathfrak{S}, \mathcal{B}_2)) \implies (\mathfrak{L}(\mathfrak{S}, \mathcal{B}_1) > \mathfrak{L}(\mathfrak{S}, \mathcal{B}_2))$$

Fig. 11. Key auxiliary definitions for inference rules.

standard condition in rely/guarantee reasoning. $\mathsf{wffLay}(\mathfrak{L})$ (Fig. 11) asks if the two base definite actions have the same layer feature, their layers must be equal. So when we pick a subset $\mathcal{D}'$ of $\mathcal{D}$ in $\mathcal{D}' \leqslant \mathcal{D}$, we know the subset have the same layers because they have the same layer features. $\mathsf{wffAct}(R, \mathcal{D})$ (Fig. 11) restricts how an enabled definite action is finished. It says environment non-delaying steps preserve an enabled definite action. Together with $G_\mathtt{t} \Rightarrow R_{\mathtt{t}'}$, the only way to disable it is to let thread t fulfill the definite action itself. $(P \Rightarrow \neg\mathsf{Enabled}(\mathcal{D}))$ says that when the thread terminates, there are no enabled definite actions.

The WHL rule. The verification of the loop body uses a precondition p', implied by the loop invariant p and the loop condition B. If the loop is not blocked, represented by an assertion Q, a $\Diamond$-token needs to be consumed to start a round of loop. So the loop will terminate if Q is continuously true and all $\Diamond$-tokens are exhausted. If the loop is blocked, i.e., Q does not hold, we use the definite progress condition introduced by LiLi to enforce that Q will eventually always hold.

Definition 4.1 (Definite Progress). $\mathfrak{S} \models (R, G\colon \mathcal{D} \xrightarrow{f} Q)$ iff the following hold for any t:

(1) either $\mathfrak{S} \models Q_\mathtt{t}$, or there exists $\mathtt{t}'$ such that $\mathtt{t}' \neq \mathtt{t}$ and $\mathfrak{S} \models \mathsf{Enabled}(\mathcal{D}_{\mathtt{t}'})$;
(2) for any $\mathtt{t}' \neq \mathtt{t}$, $\mathcal{B}$ and $\mathfrak{S}'$, if $\mathcal{B} \leqslant \mathcal{D}$ and $(\mathfrak{S}, \mathfrak{S}', 0) \models R_\mathtt{t} \wedge \langle \mathcal{B}_{\mathtt{t}'} \rangle$, then $f_\mathtt{t}(\mathfrak{S}') < f_\mathtt{t}(\mathfrak{S})$;
(3) for any $\mathfrak{S}'$, if $(\mathfrak{S}, \mathfrak{S}', 0) \models R_\mathtt{t} \vee G_\mathtt{t}$, then $f_\mathtt{t}(\mathfrak{S}') \leq f_\mathtt{t}(\mathfrak{S})$.

f is a function that maps the relational memory $\mathfrak{S}$ to some metrics with a well-founded order $<$.

The three conditions require the following. (1) A blocked thread must wait for enabled definite actions of other threads to happen. (2) Once these threads fulfill their definite actions, a metric of the current thread will decrease. (3) The metric does not increase at all steps. Note that we allow delaying actions to increase the metric because delaying actions would consume ⧫-tokens. The effect of delaying actions is in ATOM rule, which is the same as in LiLi and omitted here.

The assertion J is implied by p and is stable under all transitions (the definition of Sta is also the same as LiLi and omitted) so that it always holds during the loop. We require J to imply the definite progress to ensure the metric decreases whenever the waited definite actions happen. Definite progress is created from $\mathcal{D}'$. $\mathcal{D}' \leqslant_{J,\mathfrak{L}} \mathcal{D}$ enforces that the current thread can only depend on definite actions with stably lower layers. The definition has been given and explained in (2.2).

The LIVE-FRM rule. The LIVE-FRM rule allows to verify C with local definite actions $\mathcal{D}'$ and layer mappings $\mathfrak{L}'$ that C's progress actually relies on. Then we can know C still progresses in a different context, $\mathcal{D}$ and $\mathfrak{L}$, without redoing the proof. $\mathcal{D}''$ is the *frame* that specifies the extra definite actions in the context. J is an invariant that always holds during the execution of C (similar to the WHILE rule). layGt (Fig. 11) says that for any definite action $\mathcal{B}$ in the frame $\mathcal{D}''$, if $\mathcal{B}$ is enabled at some state $\mathfrak{S}$ satisfying J, $\mathcal{B}$ may depend on $\mathcal{D}'$ to fulfill, and therefore $\mathcal{B}$ should be at a strictly higher layer than all the definite actions in $\mathcal{D}'$.

Consider `lock B` in Fig. 1, which occurs in both `B()` and `AB()`. We can verify the code once with $\mathcal{D}_B$ in `B()`, and then apply the LIVE-FRM rule to work with $\mathcal{D}_A \wedge \mathcal{D}_B$ for `AB()`. $\mathsf{layGt}(\mathfrak{L}, \mathcal{D}_A, J, \mathcal{D}_B)$ holds because $\mathcal{D}_A$'s layer is strictly higher than $\mathcal{D}_B$ when the code of `AB()` requests for `B`.

In addition, the layer functions should satisfy $\mathfrak{L}' \leqslant \mathfrak{L}$, which requires $\mathfrak{L}$ to be an order-preserving extension of $\mathfrak{L}'$. Specifically, it includes two conditions, (1) subMap says that $\mathfrak{L}$ should assign a layer to any definite action that $\mathfrak{L}'$ has given a layer; and (2) orderPresv requires that the relative ordering of layers in $\mathfrak{L}'$ should be preserved in $\mathfrak{L}$.

$\mathfrak{L}'$ can reassign layers and work with different locks even when the locks have different layers in $\mathfrak{L}$. E.g., we can use parametric specifications ($\mathcal{D}_X$ and $\mathfrak{L}_X$) to verify `lock X` so that it is a proof for both `lock A` and `lock B`. Although $\mathfrak{L}$ assigns different layers to $\mathcal{D}_A$ and $\mathcal{D}_B$, $\mathfrak{L}_X$ only contains a mapping for $\mathcal{D}_X$ and can map $\mathcal{D}_X$ to an arbitrary layer to satisfy $\mathfrak{L}_X \leqslant \mathfrak{L}$. We will show how LIVE-FRM rule achieves modular proofs for the nested locking object (Fig. 1) in Sec. 5.

4.3 Soundness of the Logic

The logic PODD is sound for proving linearizability and starvation-freedom/deadlock-freedom of concurrent objects, as shown by the following theorem.

THEOREM 4.2 (SOUNDNESS). *If* $\mathfrak{L}, \mathcal{D}, R, G \vdash \{P\}\Pi : \Gamma$, *then*

(1) both $\Pi \preceq_P^{\mathsf{lin}} \Gamma$ *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 $\Pi \preceq_P^{\mathsf{lin}} \Gamma$ says the object Π is linearizable with respect to the atomic specification Γ if the initial states satisfy the relational assertion P. The logic guarantees $\mathsf{deadlock\text{-}free}_P(\Pi)$, and if all actions are level 0, the logic further proves $\mathsf{starvation\text{-}free}_P(\Pi)$. The formal definitions and proofs are omitted here and can be found in the Appendix.

Note that PODD's proof can be built from LiLi's proof. We can lift LiLi's definite action to PODD's definite action (by setting $\mathcal{F}$ to an arbitrary value). The layer function maps all definite actions in all states to a constant. For each logic rule in PODD, the extra conditions (not in LiLi's corresponding rule) trivially hold. Moreover, LiLi can be applied to verify non-blocking objects (i.e., wait-freedom and lock-freedom), which means PODD can also support them.

5 EXAMPLES

We have applied PODD to verify all examples verified by LiLi. They include objects with coarse-grained, fine-grained, and optimistic synchronization. With layered definite actions, we can specify a parametric definite action uniformly for each lock. The specification for each lock statement only talks about locally-needed definite actions and layer mappings so the internal progress proof of each lock can be reused. PODD can also soundly handle dynamic dependencies of progress through dynamic layers. As case studies, we verify the nested locking object and the `move` method of transfer lists. We have also proved a practical concurrent file system that exhibits dynamic dependencies of progress as introduced in Sec. 2.3.1 (the result is published in a separate work). The verification of these cases illustrates the wide applicability of PODD. To the best of our knowledge, we are the first to modularly verify the progress of transfer lists.

5.1 Nested Locking Object

We define the assertions and specifications for the nested locking object in Fig. 12 and verify `AB()` method in Fig. 13. For `AB()`, its abstract atomic operation can be seen as **skip** and omitted here. We choose to implement the locks as test-and-set locks. Because a test-and-set lock allows a thread to delay another thread by first acquiring the lock, the object is deadlock-freedom. In Appendix, we also implement the lock as ticket locks to ensure starvation-freedom.

$\mathsf{lock}_s(\mathsf{x}) \stackrel{\text{def}}{=} (\mathsf{x} = s)$ $\quad \mathsf{lock}(\mathsf{x}) \stackrel{\text{def}}{=} \exists s.\, \mathsf{lock}_s(\mathsf{x})$ $\quad \mathsf{lockedBy}_\mathsf{t}(\mathsf{x}) \stackrel{\text{def}}{=} \mathsf{lock}_\mathsf{t}(\mathsf{x}) \wedge (\mathsf{t} \neq 0)$

$\mathsf{unlocked}(\mathsf{x}) \stackrel{\text{def}}{=} \mathsf{lock}_0(\mathsf{x})$ $\quad \mathsf{locked}(\mathsf{x}) \stackrel{\text{def}}{=} \exists \mathsf{t}.\, \mathsf{lockedBy}_\mathsf{t}(\mathsf{x})$ $\quad \mathsf{notOwn}_\mathsf{t}(\mathsf{x}) \stackrel{\text{def}}{=} \exists \mathsf{t}'.\, \mathsf{lock}_{\mathsf{t}'}(\mathsf{x}) \wedge (\mathsf{t} \neq \mathsf{t}')$

$R_\mathsf{t} \stackrel{\text{def}}{=} \bigvee_{\mathsf{t}' \neq \mathsf{t}} G_{\mathsf{t}'}$ $\quad G_\mathsf{t} \stackrel{\text{def}}{=} G_\mathsf{t}(\mathsf{L}) \vee G_\mathsf{t}(\mathsf{A}) \vee G_\mathsf{t}(\mathsf{B}) \vee \mathit{SetStr}_\mathsf{t}$ $\quad G_\mathsf{t}(x) \stackrel{\text{def}}{=} \mathit{Acq}_\mathsf{t}(x) \vee \mathit{Rel}_\mathsf{t}(x)$

$\mathit{Acq}_\mathsf{t}(x) \stackrel{\text{def}}{=} (\mathsf{unlocked}(\mathsf{x}) \ltimes_1 \mathsf{lockedBy}_\mathsf{t}(\mathsf{x})) * \mathsf{Id}$ $\quad \mathit{Rel}_\mathsf{t}(x) \stackrel{\text{def}}{=} (\mathsf{lockedBy}_\mathsf{t}(\mathsf{x}) \ltimes_0 \mathsf{unlocked}(\mathsf{x})) * \mathsf{Id}$

$\mathit{SetStr}_\mathsf{t} \stackrel{\text{def}}{=} ((\mathsf{lockedBy}_\mathsf{t}(\mathsf{L}) * \mathsf{str}\_) \ltimes_0 (\mathsf{lockedBy}_\mathsf{t}(\mathsf{L}) * \mathsf{str}\_)) * \mathsf{Id}$

$\mathcal{D}_\mathsf{t} \stackrel{\text{def}}{=} \mathcal{B}_\mathsf{t}(\mathsf{L}) \wedge \mathcal{B}_\mathsf{t}(\mathsf{A}) \wedge \mathcal{B}_\mathsf{t}(\mathsf{B}) \wedge \mathcal{B}_0$

$\mathit{LayFeat} \stackrel{\text{def}}{=} \mathit{Var} \mid \bot$

$\mathcal{B}_\mathsf{t}(\mathsf{x}) \stackrel{\text{def}}{=} (\mathsf{bp}_\mathsf{t}(\mathsf{x}) \rightsquigarrow \mathsf{bq}(\mathsf{x}), \mathsf{x})$

$\mathsf{bp}_\mathsf{t}(\mathsf{x}) \stackrel{\text{def}}{=} \mathsf{lockedBy}_\mathsf{t}(\mathsf{x}) * \textbf{true}$

$\mathsf{bq}(\mathsf{x}) \stackrel{\text{def}}{=} \mathsf{unlocked}(\mathsf{x}) * \textbf{true}$

$\mathcal{B}_0 \stackrel{\text{def}}{=} (\mathsf{False} \rightsquigarrow \mathsf{True}, \bot)$

$\mathit{Layer} \stackrel{\text{def}}{=} \mathit{Nat}$

$$\mathfrak{L}(\mathfrak{S}, \mathcal{B}) = \begin{cases} 3 & \text{if } \mathcal{B}.\mathcal{F} = \mathsf{L} \\ 2 & \text{if } \mathcal{B}.\mathcal{F} = \mathsf{A} \wedge \mathfrak{S} \models \mathsf{str}=\text{"AB"} * \textbf{true} \\ 2 & \text{if } \mathcal{B}.\mathcal{F} = \mathsf{B} \wedge \mathfrak{S} \models \mathsf{str}=\text{"BA"} * \textbf{true} \\ 0 & \text{if } \mathcal{B}.\mathcal{F} = \bot \\ 1 & \text{otherwise} \end{cases}$$

Fig. 12. Assertions and specifications for the nested locking object.

$\mathfrak{L}, \mathcal{D}, R, G \vdash$
$\{\mathsf{notOwn}_\mathsf{t}(\mathsf{L}) * \mathsf{notOwn}_\mathsf{t}(\mathsf{A}) * \mathsf{notOwn}_\mathsf{t}(\mathsf{B}) * \mathsf{str}\_ \wedge \blacklozenge(3)\}$

// apply LIVE-FRM rule

1 `lock L;`
$\{\mathsf{lockedBy}_\mathsf{t}(\mathsf{L}) * \mathsf{notOwn}_\mathsf{t}(\mathsf{A}) * \mathsf{notOwn}_\mathsf{t}(\mathsf{B}) * \mathsf{str}\_ \wedge \blacklozenge(2)\}$

2 `str := "AB";`
$\{\mathsf{lockedBy}_\mathsf{t}(\mathsf{L}) * \mathsf{notOwn}_\mathsf{t}(\mathsf{A}) * \mathsf{notOwn}_\mathsf{t}(\mathsf{B}) * \mathsf{str}=\text{"AB"} \wedge \blacklozenge(2)\}$

3 `lock A;`
$\{\mathsf{lockedBy}_\mathsf{t}(\mathsf{L}) * \mathsf{lockedBy}_\mathsf{t}(\mathsf{A}) * \mathsf{notOwn}_\mathsf{t}(\mathsf{B}) * \mathsf{str}=\text{"AB"} \wedge \blacklozenge(1)\}$

4 `lock B; str := "";`
$\{\mathsf{lockedBy}_\mathsf{t}(\mathsf{L}) * \mathsf{lockedBy}_\mathsf{t}(\mathsf{A}) * \mathsf{lockedBy}_\mathsf{t}(\mathsf{B}) * \mathsf{str}=\text{""}\}$

5 `unlock L; unlock A; unlock B;`
$\{\mathsf{notOwn}_\mathsf{t}(\mathsf{L}) * \mathsf{notOwn}_\mathsf{t}(\mathsf{A}) * \mathsf{notOwn}_\mathsf{t}(\mathsf{B}) * \mathsf{str}\_\}$

Fig. 13. Proofs for the AB() method of nested locking object.

Assertions. $\mathsf{lock}_s(\mathsf{x})$ says the lock x has value s. We define other useful assertions. unlocked(x) and locked(x) say the lock x is available or locked by some thread, respectively. $\mathsf{lockedBy}_\mathsf{t}(\mathsf{x})$ and $\mathsf{notOwn}_\mathsf{t}(\mathsf{x})$ say the lock x is owned or not owned by thread t, respectively. We then define the transitions, including acquiring a lock and releasing a lock. *SetStr* will be discussed later. Note that acquiring a lock would delay other threads, so it is a level-1 action.

Definite actions and layers. We define a definite action for each lock release described by $\mathcal{B}(\mathsf{x})$. We also define a default definite action $\mathcal{B}_0$, which is never enabled. $\mathcal{B}_0$ serves as a placeholder when the command is not blocked. $\mathcal{B}(\mathsf{A})$ depends on $\mathcal{B}(\mathsf{B})$ in `AB()` method, while they have reverse dependency relation in `BA()` method. So we introduce the write-only auxiliary string `str` to distinguish the two contexts. The *SetStr* transition sets the value of `str` with lock `L` holden. We assign `"AB"` to `str` (line 2 in Fig. 13), indicating we are in `AB()` method.

The layer feature of $\mathcal{B}(\mathsf{x})$ is the *Var* x, indicating the lock. $\mathcal{B}(\mathsf{L})$ has the highest layer 3, so it can depend on the release of lock `A` and `B`. Normally, $\mathcal{B}(\mathsf{A})$ and $\mathcal{B}(\mathsf{B})$ have layer 1. However, $\mathcal{B}(\mathsf{A})$ has layer 2 when enabled in `AB()` method so that it can depend on $\mathcal{B}(\mathsf{B})$. Similarly, $\mathcal{B}(\mathsf{B})$ has layer 2 when enabled in `BA()` method. $\mathcal{B}_0$ has a special layer feature $\bot$ and lowest layer 0.

Proof for OBJ rule. The OBJ rule checks that definite actions are well-formed through $\mathsf{wffAct}(R, \mathcal{D})$. Environment non-delaying actions ($\mathit{Rel}_{\mathsf{t}'}$ and $\mathit{SetStr}_{\mathsf{t}'}$) preserve that the lock is held by thread t ($\mathsf{Enabled}(\mathcal{D}_\mathsf{t})$). The only way to fulfill it is to release the lock itself (Rel_t). Other specification checks trivially hold. The OBJ rule then asks to prove each method.

$$\mathfrak{L}_{\mathsf{X}}(\mathfrak{S},\mathcal{B}) = \begin{cases} 1 & \text{if } \mathcal{B}.\mathcal{F} = \mathsf{X} \\ 0 & \text{if } \mathcal{B}.\mathcal{F} = \bot \\ \textit{undefined} & \text{otherwise} \end{cases}$$

$\mathcal{D}' \stackrel{\text{def}}{=} \mathcal{B}(\mathsf{X})$

$Q \stackrel{\text{def}}{=} (\mathsf{unlocked}(\mathsf{X}) \vee \mathsf{lockedBy}_t(\mathsf{X})) * P$

$J \stackrel{\text{def}}{=} \mathsf{lock}(\mathsf{X}) * P$

$$f(\mathfrak{S},\mathsf{X}) = \begin{cases} 1 & \text{if } \mathfrak{S} \models \mathsf{locked}(\mathsf{X}) * \textbf{true} \\ 0 & \text{if } \mathfrak{S} \models \mathsf{unlocked}(\mathsf{X}) * \textbf{true} \end{cases}$$

$$\mathfrak{L}_0(\mathfrak{S},\mathcal{B}) = \begin{cases} 0 & \text{if } \mathcal{B}.\mathcal{F} = \bot \\ \textit{undefined} & \text{otherwise} \end{cases}$$

$\mathfrak{L}_{\mathsf{X}}, \mathcal{B}(\mathsf{X}) \wedge \mathcal{B}_0, R, \mathit{Acq}(\mathsf{X}) \vdash$
$\{(\mathsf{notOwn}_t(\mathsf{X}) \wedge \blacklozenge) * P\}$

```
6  local b := false;
```
$\{((\mathsf{notOwn}_t(\mathsf{X}) \wedge \blacklozenge \wedge \Diamond \wedge \neg \mathsf{b}) \vee (\mathsf{lockedBy}_t(\mathsf{X}) \wedge \mathsf{b})) * P\}$

```
7  while(!b) {
```
// apply LIVE-FRM rule

$\mathfrak{L}_0, \mathcal{B}_0, R, \mathit{Acq}(\mathsf{X}) \vdash$
$\{((\mathsf{unlocked}(\mathsf{X}) \wedge \blacklozenge) \vee (\mathsf{locked}(\mathsf{X}) \wedge \blacklozenge \wedge \Diamond)) * P\}$

```
8    b := cas(&X, 0, t); }
```

Fig. 14. Proof of lock X for the nested locking object.

Proof for AB() method and application of LIVE-FRM *rule.* In Fig. 13, the precondition of `AB()` says the current thread does not hold any lock, and `str` equals an arbitrary value. We need three ⧫-tokens because acquiring each lock would delay others and cost a ⧫-token. The proof for each lock statement shares the same proof sketch (i.e., proof for `lock X`).

We could first apply the LIVE-FRM rule to specify local definite actions and layer mappings needed during `lock X`. We can verify `lock X` with $\mathcal{B}(\mathsf{X}) \wedge \mathcal{B}_0$, which locally talks about the definite release of lock X and $\mathcal{B}_0$, a placeholder when no definite action is needed. We can safely discard other definite actions that may be enabled during `lock X` because layGt (Fig. 11) of LIVE-FRM rule already proves they have higher layers than $\mathcal{B}(\mathsf{X})$ and $\mathcal{B}_0$. Also, the layer function $\mathfrak{L}_{\mathsf{X}}$ can reassign the layers (see Fig. 14). We only need to ensure the layering mappings of $\mathfrak{L}_{\mathsf{X}}$ are a subset of $\mathfrak{L}$, and relative layer relations in $\mathfrak{L}_{\mathsf{X}}$ also hold in $\mathfrak{L}$ to satisfy $\mathfrak{L}_X \leqslant \mathfrak{L}$.

Proof of lock X. In Fig. 14, the precondition of `lock X` requires $\mathsf{notOwn}_t(\mathsf{X})$, saying the current thread does not own the lock, a ⧫-token, and a frame part P (discussed later). To prove the **while**-loop in `lock X`, we define the $\mathcal{D}'$, J, Q, and f. $\mathcal{D}'$ specifies the definite release action of lock X. Q says the lock is available or holden by the current thread. J defines that the thread requests for the lock. The metric $f(\mathfrak{S},\mathsf{X})$ is determined by whether X is locked or unlocked.

The proof of the definite progress $J \Rightarrow (R, \mathit{Acq}(\mathsf{X}) : \mathcal{B}(\mathsf{X}) \xrightarrow{f} Q)$ exhibits local reasoning.

(1) Either lock X is unlocked, or some thread has acquired X.
(2) If some thread releases lock X, then the metric decreases.
(3) Non-delaying actions will not increase the metric. Either a non-delaying action (*Rel* on other locks and *SetStr*) keeps X unchanged, where the metric does not change, or the non-delaying action releases X (*Rel* on X), where the metric decrease.

The above proof and the proof of $\mathcal{B}(\mathsf{X}) \leqslant_{J,\mathfrak{L}_X} \mathcal{B}(\mathsf{X}) \wedge \mathcal{B}_0$ can be reused for each `lock` statement in nested locking object. When proving the command inside the **while**-loop, we can further apply the LIVE-FRM rule to strip away $\mathcal{B}(\mathsf{X})$, showing that no definite action is needed for the command. The layer function can also be further simplified to $\mathfrak{L}_0$ (see Fig. 14). In fact, all progress proofs for `lock X` are reusable.

The previous discussion has left out some details. E.g., the verification of `lock X` uses a strengthened guarantee condition $\mathit{Acq}(\mathsf{X})$ due to CSQ rule, and HIDE-◇ rule introduces the ◇-token in the **while**-loop invariant of `lock X`. These rules can be found in the Appendix.

Discussion on modularity. Still, there is room for more modular proofs. In verifying `lock X`, the judgment takes the object's rely condition R, so the proof may not be reused across different objects. Also, the frame P may be instantiated differently for verifying different locks, e.g., P for `lock L`

```
struct Node {
  int lock;
  int data;
  struct Node *next;
}
struct List {
  struct Node *Head;
  int level;
}
```

```
initialize(){
  l1.Head := cons(0, GUARD, null);
  l1.Head.next := cons(0, GUARD, null);
  l2.Head := cons(0, GUARD, null);
  l2.Head.next := cons(0, GUARD, null);
  l1.level := 1;  l2.level := 1;
  mutex := 0;
}
```

```
move12():
    local x, y, z, e, m, n, u;
 1  lock(mutex);
 2  x := l1.head;
 3  lock(x);
 4  y := x.next;
 5  e := y.data;
 6  if (e = GUARD) {
      // l1 is empty
 7    unlock(mutex);
 8    unlock(x);
 9    return false;
10  }
11  l2.level := 0;
12  (m, n) := locate(l2, e);
13  u := n.data;
14  if (u = e) {
      // if e already in l2
15    unlock(x);
16    unlock(m);
17    l2.level := 1;
18    unlock(mutex);
19    return false;
20  }
    // remove y in l1
21  l2.level := 2;
22  lock(y);
23  z := y.next;
24  x.next := z;
    // insert y at the end of l2
25  y.next := m.next;
26  m.next := y;
27  unlock(x);
28  unlock(y);
29  unlock(m);
30  l2.level := 1;
31  unlock(mutex);
32  return true;
```

Fig. 15. Transfer lists with auxilary code.

specifies that the current thread does not hold lock A and B, str equals an arbitrary value, and there are two ⧫-tokens. To separate the shared memory in P, we need to support spatial separation on R, G, and $\mathcal{D}$, which is currently not supported. More specifically, most proofs do not need to peek into what P says. P only needs to be opened and proved stable under R. The good news is that the stability proof for P is done once and for all. In addition, we can provide a FRM rule to frame thread-local states as shown in the full logic rules of the Appendix.

5.2 Transfer Lists

Fig. 15 shows the code of a simplified transfer lists object, where we assume only two lists, l1 and l2. The transfer lists object is starvation-freedom or deadlock-freedom, depending on the lock used. We can verify that each operation (add, rmv, and move) refines an abstract atomic operation. As discussed in Sec. 2, the major challenge is that this example shows dynamic dependencies of progress. Here we focus on how we handle the challenge by verifying move. Specially, we will explain the specifications of definite actions and layers, which account for the dynamic dependencies. Full specifications and proofs can be found in Appendix.

Implementation and auxiliary code. The $\texttt{move}_{12}()$ method transfers a node from list l1 to l2, and there is also a $\texttt{move}_{21}()$ method (the use of subscript here is for simplicity). The implementation is mostly the same as Fig. 5 (ignore the code in red).

If the implementation tries to acquire the lock of l2's node when holding the lock of l1's node, the definite lock release of l1's node may depend on the definite lock release of l2's node. For

$$\mathcal{D}_t \stackrel{\text{def}}{=} \mathcal{B}1_t \wedge (\forall x.\, \mathcal{B}2_t(x)) \wedge \mathcal{B}_0 \qquad \mathit{LayFeat} \stackrel{\text{def}}{=} \mathit{Addr} \mid \mathcal{M} \mid \bot \qquad \mathcal{B}_0 \stackrel{\text{def}}{=} (\mathsf{False} \rightsquigarrow \mathsf{True}, \bot)$$

$$\mathcal{B}1_t \stackrel{\text{def}}{=} (\mathsf{bp1}_t \rightsquigarrow \mathsf{bq1}, \mathcal{M}) \qquad \mathsf{bp1}_t \stackrel{\text{def}}{=} \mathsf{mutex\_locked}_t * \mathbf{true} \qquad \mathsf{bq1} \stackrel{\text{def}}{=} \mathsf{mutex\_unlocked} * \mathbf{true}$$

$$\mathcal{B}2_t(x) \stackrel{\text{def}}{=} (\mathsf{bp2}_t(x) \rightsquigarrow \mathsf{bq2}(x), x) \qquad \mathsf{bp2}_t(x) \stackrel{\text{def}}{=} \mathsf{locked}_t(x) * \mathbf{true} \qquad \mathsf{bq2}(x) \stackrel{\text{def}}{=} \mathsf{unlocked}(x) * \mathbf{true}$$

$$\mathit{Layer} \stackrel{\text{def}}{=} \mathit{Nat} \times \mathit{Nat} \qquad (n_1, n_2) < (n_1', n_2') \text{ iff } n_1 < n_1' \vee (n_1 = n_1' \wedge n_2 < n_2')$$

$$\mathfrak{L}(\mathfrak{S}, \mathcal{B}) = \begin{cases} (2,0) & \text{if } \mathcal{B}.\mathcal{F} = \mathcal{M} \\ (n, \mathsf{len(B)}) & \text{if } \mathcal{B}.\mathcal{F} = x \wedge \mathfrak{S} \models P \\ (0,0) & \text{otherwise} \end{cases} \qquad P \stackrel{\text{def}}{=} \exists L, L', \mathsf{l}, \mathsf{A}, \mathsf{B}, n.\ \mathsf{ls}_L(\mathsf{l.Head}, \mathsf{A}, x) * \mathsf{ls}_{L'}(x, \mathsf{B}, \mathtt{null}) * (\mathsf{l.level} = n) * \mathbf{true}$$

Fig. 16. Definite actions and layers of the transfer lists.

$$\mathsf{mutex\_locked}_t \stackrel{\text{def}}{=} (\mathtt{mutex} = t) \wedge (t \neq 0) \qquad \mathsf{mutex\_unlocked} \stackrel{\text{def}}{=} (\mathtt{mutex} = 0) \qquad \mathsf{unlocked}(p) \stackrel{\text{def}}{=} (p.\mathtt{lock} = 0)$$

$$\mathsf{locked}_t(p) \stackrel{\text{def}}{=} (p.\mathtt{lock} = t) \wedge (t \neq 0) \qquad \mathsf{lock}_s(p) \stackrel{\text{def}}{=} \mathsf{locked}_s(p) \vee (s = 0 \wedge \mathsf{unlocked}(p))$$

$$\mathsf{ls}_L(x, A, y) \stackrel{\text{def}}{=} (x = y \wedge A = \epsilon \wedge L = \epsilon \wedge \mathsf{emp}) \vee (x \neq y \wedge \exists z, v, A', s, L'.\ A = v :: A' \wedge L = s :: L' \wedge \mathsf{N}_s(x, v, z) * \mathsf{ls}_{L'}(z, A', y))$$

$$\mathsf{N}_s(p, v, y) \stackrel{\text{def}}{=} \mathsf{lock}_s(p) * (p.\mathtt{data} = v) * (p.\mathtt{next} = y) \qquad \mathsf{len}(\epsilon) \stackrel{\text{def}}{=} 0 \qquad \mathsf{len}(v :: A) \stackrel{\text{def}}{=} \mathsf{len}(A) + 1$$

Fig. 17. Auxilary definitions for transfer lists.

convenience, we refer to the above scenario as "list l1 depends on list l2". $\mathtt{move}_{12}()$ may first traverse l2 with l1's lock in hand, so l1 depends on l2. However, $\mathtt{move}_{21}()$ may traverse in reverse order, causing l2 to depend on l1. To account for the dynamic dependencies between the lists, we introduce a write-only auxiliary field `level` in the `struct List` (Fig. 15). The levels of both lists are initialized to 1, meaning they are at the same level without dependencies. Before the $\mathtt{move}_{12}()$ method tries to traverse l2, we should lower its level to 0 (line 11), so l1 with level 1 can depend on l2. When the `move` finds that l2 does not have the value after line 20, it needs to acquire the lock of y in l1 at line 22. This lock acquisition requires the level of l2 to be higher than l1. So we lift l2's level to 2 at line 21 to allow the dependency. After the operation fails or finishes, we reset l2's level to 1 (line 17 or 30).

Definite actions and layers. Fig. 16 shows the definite actions and layers for the transfer lists. $\mathcal{B}1$ expresses the definite release of the lock `mutex`. $\mathcal{B}2(x)$ specifies that the lock for each node x will always be released. $\mathcal{B}_0$ is the placeholder when no definite action is needed. The layer features are either the address *Addr* of the node, a special sign $\mathcal{M}$ for the lock `mutex`, or $\bot$ for $\mathcal{B}_0$.

We instantiate the *Layer* to a pair of *Nat* and define the order between two layers following the alphabetical order. The first field of a layer corresponds to the `level` field of the list, which accounts for the dynamic layer relations between list l1 and l2. The second field refers to the distance between a node and the list's end, specifying the layer relations inside a list. For the definite action $\mathcal{B}1$, we give a default highest layer $(2, 0)$, allowing it to depend on other definite actions. For a definite action $\mathcal{B}2(x)$, which specifies the lock release on node x, we should first locate the node in a list l expressed by $\mathsf{ls}_L(\mathsf{l.Head}, \mathsf{A}, x)$ (defined in Fig. 17) to know the level n of l. We then get the part of the list from the node to the end of the list, specified by $\mathsf{ls}_{L'}(x, \mathsf{B}, \mathtt{null})$. The length $\mathsf{len}(\mathsf{B})$ gives the distance between the node and the end.

The dynamic layers concisely specify the dependencies among definite actions. For definite actions in the same list, since their first layer fields are the same, the second fields give their layer relations. Moreover, the second field matches the intuition that their layers for the nodes should be decreasing starting from the list head. For definite actions in different lists, the first fields give their dependencies. Because the first field corresponds to the auxiliary state that dynamically changes during code execution, it effectively accounts for the dynamic dependencies between lists.

6 RELATED WORK AND CONCLUSION

There has been much work on verifying progress properties for concurrent programs. Below we focus on the most closely related work. Many efforts [Balzer et al. 2019; Hamin and Jacobs 2018; Jacobs et al. 2022; Leino et al. 2010] support deadlock-freedom verification. They track dependencies between primitive blocking constructs, e.g., lock primitive, and ensure the lack of circular blocking. Their dependencies between blocking events are used to detect deadlocks, which is more of a safety property. Our dependencies between definite actions are used to enforce termination, a progress property. Therefore, they cannot apply for ad-hoc synchronization, like busy-waiting patterns in fine-grained programs, which are supported by definite actions.

Gu et al. [Kim et al. 2017] verify the progress of the MCS lock by reasoning about abstract logs of events, which is expensive. Also, their approach provides less guidance on how to prove blocking objects, e.g., how to verify cases with dynamic dependencies of progress.

The program logic LiLi [Liang and Feng 2016] can verify starvation-freedom and deadlock-freedom of blocking objects. They propose several ideas that inspire us, including using definite actions for blocking and stratified tokens to support delays and rollbacks for deadlock-free objects. An effort [Zou et al. 2022] uses LiLi's theory to prove the termination of a concurrent file system, AtomFS [Zou et al. 2019]. AtomFS has a `rename` method that may move a subtree of nodes from the source directory to the target directory similar to the `move` method of transfer lists. However, their proofs use LiLi's strict definite actions to unfold and expose full dependency chains, which are not modular. The problem is addressed in this paper by structuring definite actions into layers.

The follow-up work of LiLi [Liang and Feng 2018] studies objects with partial methods, i.e., methods that are supposed not to return under certain circumstances. Typical examples include various lock implementations, e.g., test-and-set locks and ticket locks. We conjecture that the support of partial methods is compatible with our layered definite actions and can be integrated into our logic if needed.

TaDA Live [D'Osualdo et al. 2021] aims at total correctness, in particular, verifying the termination of programs. It proposes obligations, a new ghost state that encodes liveness invariants. The obligations are statically layered so TaDA Live can handle nested dependencies by arranging termination proof in a hierarchy. In their verification of lock coupling list, to work around its limitation of static layers, TaDA-Live introduces techniques such as parametric specifications, which result in less concise and intuitive proofs. For dynamic dependencies, TaDA-Live claims "not aware of any example that critically requires more expressive layers" and "not clear how to ensure soundness if interference on layers is allowed". By contrast, our work finds cases with dynamic dependencies of progress, and PODD naturally supports them with the dynamic layering of definite actions. Furthermore, PODD avoids circular dependencies among definite actions through dependency enforcement.

Substantial efforts use temporal logic for liveness verification. Temporal reasoning directly works with traces to express progress properties in a unified and general way. However, it lacks guidance for how to discharge proof obligations. PODD's layered definite actions allow one to understand the progress of concurrent objects and construct proofs correspondingly.

Conclusion. We have studied the verification of blocking objects in three aspects. First, we propose dynamically layered definite actions to modularly verify nested and dynamic dependencies of progress. Second, we develop a program logic PODD, which is sound for proving linearizability and starvation-freedom/deadlock-freedom of concurrent objects. Third, we apply PODD to successfully verify objects with nested and dynamic dependencies of progress. We have also applied PODD to a concurrent file system (published separately), which shows the wide applicability of PODD. In the future, we hope to mechanize PODD and develop tools to automate the verfication process.

.

REFERENCES

Stephanie Balzer, Bernardo Toninho, and Frank Pfenning. 2019. Manifest Deadlock-Freedom for Shared Session Types.. In *ESOP*. 611–639.

Linux documentation. 2022. Kernel subsystem documentation » Filesystems in the Linux kernel » Directory Locking. https://www.kernel.org/doc/html/latest/filesystems/directory-locking.html

Emanuele D'Osualdo, Julian Sutherland, Azadeh Farzan, and Philippa Gardner. 2021. TaDA Live: Compositional reasoning for termination of fine-grained concurrent programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 43, 4 (2021), 1–134.

Xinyu Feng. 2009. Local rely-guarantee reasoning. In *POPL*. 315–327.

Ivana Filipović, Peter O'Hearn, Noam Rinetzky, and Hongseok Yang. 2010. Abstraction for concurrent objects. *Theor. Comput. Sci.* 411, 51-52 (2010), 4379–4398.

Linux github. 2022. The Linux Code of Lock Acquisitions for Rename. https://github.com/torvalds/linux/blob/master/fs/namei.c

Jafar Hamin and Bart Jacobs. 2018. Deadlock-free monitors. In *European Symposium on Programming*. Springer, 415–441.

Maurice Herlihy and Nir Shavit. 2008. *The Art of Multiprocessor Programming*. Morgan Kaufmann.

Maurice Herlihy and Nir Shavit. 2011. On the Nature of Progress. In *Proceedings of the 15th International Conference on Principles of Distributed Systems (OPODIS 2011)*. 313–328.

Maurice Herlihy and Jeannette Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (1990), 463–492.

Jules Jacobs, Stephanie Balzer, and Robbert Krebbers. 2022. Connectivity graphs: a method for proving deadlock freedom based on separation logic. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–33.

Cliff B. Jones. 1983. Tentative Steps Toward a Development Method for Interfering Programs. *ACM Trans. Program. Lang. Syst.* 5, 4 (1983), 596–619.

Jieung Kim, Vilhelm Sjöberg, Ronghui Gu, and Zhong Shao. 2017. Safety and Liveness of MCS Lock—Layer by Layer. In *Asian Symposium on Programming Languages and Systems*. Springer, 273–297.

K. Rustan M. Leino, Peter Müller, and Jan Smans. 2010. Deadlock-Free Channels and Locks. In *ESOP*. 407–426.

Hongjin Liang and Xinyu Feng. 2013. Modular Verification of Linearizability with Non-Fixed Linearization Points. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2013)*. 459–470.

Hongjin Liang and Xinyu Feng. 2016. A Program Logic for Concurrent Objects Under Fair Scheduling. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2016)*. 385–399.

Hongjin Liang and Xinyu Feng. 2018. Progress of Concurrent Objects with Partial Methods. *Proc. ACM Program. Lang.* 2, POPL, Article 20 (dec 2018), 31 pages. https://doi.org/10.1145/3158108

Hongjin Liang, Xinyu Feng, and Zhong Shao. 2014. Compositional Verification of Termination-preserving Refinement of Concurrent Programs. In *Proceedings of the Joint Meeting of the 23rd EACSL Annual Conference on Computer Science Logic and the 29th Annual ACM/IEEE Symposium on Logic in Computer Science (CSL-LICS 2014)*. 65:1–65:10.

Hongjin Liang, Jan Hoffmann, Xinyu Feng, and Zhong Shao. 2013. Characterizing Progress Properties of Concurrent Objects via Contextual Refinements. In *Proceedings of the 24th International Conference on Concurrency Theory (CONCUR 2013)*. 227–241.

Maged M. Michael and Michael L. Scott. 1996. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC*. 267–275.

Sujin Park, Irina Calciu, Taesoo Kim, and Sanidhya Kashyap. 2021. Contextual concurrency control. In *Proceedings of the Workshop on Hot Topics in Operating Systems*. 167–174.

Matthew Parkinson, Richard Bornat, and Cristiano Calcagno. 2006. Variables as Resource in Hoare Logics. In *LICS*. 137–146.

Viktor Vafeiadis. 2008. *Modular fine-grained concurrency verification*. Technical Report. PhD Thesis.

Mo Zou, Haoran Ding, Dong Du, Ming Fu, Ronghui Gu, and Haibo Chen. 2019. Using concurrent relational logic with helpers for verifying the AtomFS file system. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 259–274.

Mo Zou, Haotong Xie, Zhuoran Wei, and Haibo Chen. 2022. Verification of Termination for File System Based on Lock Coupling Traversal. *Ruan Jian Xue Bao/Journal of Software* 33 (2022), 2980–2994.

$$
\begin{array}{rrcl}
(\mathit{ThrdID}) & \mathrm{t} & \in & \mathit{Nat} \\
(\mathit{Heap}) & h, \mathbb{h} & \in & \mathit{Nat} \rightharpoonup \mathit{Int} \\
(\mathit{RelMem}) & \mathfrak{S} & ::= & (\sigma, \Sigma) \\
(\mathit{ThrdPool}) & \mathcal{K}, \mathbb{K} & ::= & \{\mathrm{t}_1 \rightsquigarrow \kappa_1, \ldots, \mathrm{t}_n \rightsquigarrow \kappa_n\} \\
(\mathit{ExecCtxt}) & \mathbf{E} & ::= & [\,] \mid \mathbf{E}; C \\
(\mathit{Evt}) & e & ::= & (\mathrm{t}, f, n) \mid (\mathrm{t}, \mathbf{ret}, n) \mid (\mathrm{t}, \mathbf{obj}) \mid (\mathrm{t}, \mathbf{clt}) \mid (\mathrm{t}, \mathbf{out}, n) \\
 & & & \mid (\mathrm{t}, \mathbf{term}) \mid (\mathbf{spawn}, n) \mid (\mathrm{t}, \mathbf{obj}, \mathbf{abort}) \mid (\mathrm{t}, \mathbf{clt}, \mathbf{abort}) \\
(\mathit{ETrace}) & \mathcal{E}, \mathbb{T} & ::= & \epsilon \mid e :: \mathcal{E} \qquad \text{(co-inductive)}
\end{array}
$$

$$
\begin{array}{rrcl}
(\mathit{Store}) & s, \mathbb{s} & \in & \mathit{Var} \rightharpoonup \mathit{Int} \\
(\mathit{Mem}) & \sigma, \Sigma & ::= & (s, h) \\
(\mathit{CallStk}) & \kappa, \mathbb{k} & ::= & \circ \mid (s_l, x, C) \\
(\mathit{PState}) & \mathcal{S}, \mathbb{S} & ::= & (\sigma_c, \sigma_o, \mathcal{K})
\end{array}
$$

Fig. 18. States and event traces.

A THE LRG-STYLE FULL VERSION OF PODD

The full version of PODD extends the advanced Rely-Guarantee-based logic LRG [Feng 2009] to support dynamic allocation and ownership transfer. The top level judgment is now in the form of $\mathfrak{L}, \mathcal{D}, R, G, I \vdash \{P\}\Pi : \Gamma$. Here the fence I is used to determine the boundary of the shared memory following LRG [Feng 2009]. Just like P, I is also a relational assertion specifying the consistency relation between the concrete data representation and the abstract value.

A.1 Language, states and operational semantics

The language presented in Sec. 3 does not show the **print**(E) and **end** command. The **print**(E) command generates externally observable events, which are used to observe program behaviors. **end** is an auxiliary command appended at the end of each thread, which generates an event marking the termination of the thread.

Fig. 18 shows the program state $\mathcal{S}$, which separates the states (σ_c) accessed by clients and states (σ_o) accessed by object methods, so client code cannot touch the object data. The states σ consist of a store s that maps variable names to values and a heap h that maps addresses to values. Program state $\mathcal{S}$ also includes a thread pool, which maps each thread to its call stack. The call stack is either empty $\circ$ (i.e. the code is not executing method calls) or contains s_l for the stack of local variables, x for return value and C for the remaining client code to be executed after the method call. Similarly, there is another set of symbols for the state definitions at the high level (e.g., $\mathbb{S}$ for the program state at the high level).

Fig. 19 shows the operational semantics, which are defined by three levels of transitions. Program transitions are defined via thread transitions. Each thread transition generates an event and is lifted from the local thread transition. The local thread transition describes a step inside or outside the object method. Note that $(C, \sigma) \longrightarrow\!\!\triangleright_{\mathrm{t}}^{\omega} \cdot$ models when the atomic execution of C is blocked.

Next we illustrate the events (defined in Fig. 18) generated during the thread transitions. (t, f, n) and $(\mathrm{t}, \mathbf{ret}, n)$ show the successful execution of method call and return respectively. $(\mathrm{t}, \mathbf{out}, n)$ is generated by **print**(E) command, which is only allowed in client code. $(\mathrm{t}, \mathbf{term})$ marks the termination of the thread. Each step inside the object method generates an $(\mathrm{t}, \mathbf{obj})$ event, or an $(\mathrm{t}, \mathbf{obj}, \mathbf{abort})$ event when the thread aborts. $(\mathrm{t}, \mathbf{clt}, \mathbf{abort})$ and $(\mathrm{t}, \mathbf{clt})$ are generated similarly for the steps outside the object method. $\mathcal{E}$ represents a (possibly infinite) sequence of events and we insert a $(\mathbf{spawn}, n)$ at the beginning of $\mathcal{E}$, which helps recording the number of threads and is used to define fair executions. We treat $(\mathrm{t}, \mathbf{out}, n)$, $(\mathrm{t}, \mathbf{obj}, \mathbf{abort})$ and $(\mathrm{t}, \mathbf{clt}, \mathbf{abort})$ as externally observable events, and use get_obsv$(\mathcal{E})$ to get the subsequence of observable events in $\mathcal{E}$.

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

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

(a) program transitions

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

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

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

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

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

$$\frac{}{(\textbf{end}, (\sigma_c, \sigma_o, \circ)) \xrightarrow{(\mathrm{t}, \textbf{term})}_{\mathrm{t},\Pi} (\textbf{skip}, (\sigma_c, \sigma_o, \circ))}$$

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

$$\frac{(C, (s_o \uplus s_l, h_o)) \longrightarrow_{\mathrm{t}} \textbf{abort}}{(C, (\sigma_c, (s_o, h_o), (s_l, x, C_1))) \xrightarrow{(\mathrm{t}, \textbf{obj}, \textbf{abort})}_{\mathrm{t},\Pi} \textbf{abort}} \qquad \frac{(C, \sigma_c) \longrightarrow_{\mathrm{t}} \textbf{abort}}{(C, (\sigma_c, \sigma_o, \circ)) \xrightarrow{(\mathrm{t}, \textbf{clt}, \textbf{abort})}_{\mathrm{t},\Pi} \textbf{abort}}$$

(b) thread transitions

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

(c) local thread transitions

Fig. 19. Selected operational semantics rules.

A.2 Assertions

The full set of the syntax of the assertions is given in Fig. 20. We highlight the constructs which are not shown in Fig. 7. Following LRG [Feng 2009], we treat program variables as resources [Parkinson et al. 2006] and use own(x) for the ownership of the program variable x. We use $\lfloor p \rfloor_{\Diamond}$ to ignore the descriptions in p about the number of $\Diamond$-tokens. $\lfloor p \rfloor_{\Diamond}$ will be useful when we want to hide the

$\Diamond$-tokens that are introduced for the proofs of **while**-loops. Similarly, $\lfloor p \rfloor_{\mathsf{a}}$ ignores the descriptions in p about $\mathsf{arem}(C)$. It will be used in the inference rule for **return** commands.

$$
\begin{array}{rrcl}
(\mathit{RelAssn}) & P,Q,I & ::= & B \mid \boxed{\mathsf{own}(x)} \mid \mathsf{emp} \mid E \mapsto E \mid E \Mapsto E \mid \Vert p \Vert \mid P * Q \mid P \wedge Q \mid P \vee Q \mid \ldots \\
(\mathit{FullAssn}) & p,q,J & ::= & P \mid \mathsf{arem}(C) \mid \Diamond(E) \mid \blacklozenge(E_k,\ldots,E_1) \mid \boxed{\lfloor p \rfloor_{\Diamond}} \mid \boxed{\lfloor p \rfloor_{\mathsf{a}}} \mid p * q \mid p \wedge q \mid \ldots \\
(\mathit{RelAct}) & R,G & ::= & P \ltimes_k Q \mid [P] \mid \lfloor G \rfloor_0 \mid \mathcal{D} \mid G * G \mid G \wedge G \mid G \vee G \mid \ldots \\
(\mathit{LayFeat}) & \mathcal{F} & ::= & \ldots \qquad (\mathit{BaseDAct}) \quad \mathcal{B} ::= (P \rightsquigarrow Q, \mathcal{F}) \\
(\mathit{DAct}) & \mathcal{D} & ::= & \mathcal{B} \mid \forall x.\, \mathcal{D} \mid \mathcal{D} \wedge \mathcal{D} \\
(\mathit{Layer}) & \mathtt{l} & \in & \ldots \\
(\mathit{LayFun}) & \mathfrak{L} & \in & \mathit{RelMem} \to \mathit{BaseDAct} \rightharpoonup \mathit{Layer}
\end{array}
$$

Fig. 20. Syntax of assertions. (We highlight the constructs which are new here if compared with Fig. 7.)

Fig. 21 shows the semantics of assertions. The semantics of definite actions are the same as in Fig. 9. Fig. 22 defines the transition levels and the ordering over token numbers. Fig. 23 shows the definitions of stability and "view shifts" (which execute the abstract code). These definitions are the same as in LiLi [Liang and Feng 2016]. The syntactic sugars Id, Emp and True represent arbitrary identity transitions, empty transitions and arbitrary transitions respectively.

Fence. Since we logically split states into local and shared parts as in LRG [Feng 2009], we need a precise invariant I to uniquely determine the boundary between local and shared resources. We define the fence $I \triangleright G$ in Fig. 21(c), which says that the transition G must be made within the boundary specified by I. Here $\mathsf{Precise}(I)$ follows its usual meaning as in separation logic but is now interpreted over relational states. The need of fence $I \triangleright \{R, G\}$ is inherited from LRG. It is orthogonal to the problems studied in this paper, and readers who are unfamiliar with LRG can safely ignore it.

A.3 Inference rules

Figure 24 presents the complete set of inference rules. We have explained the OBJ, WHL and LIVE-FRM rules in Sec. 4. The slight modifications here are only to distinguish shared and local state assertions. For instance, R and G should describe transitions over shared states only, and be fenced by I.

The ATOM rule allows us to logically execute the abstract code simultaneously with every concrete step. 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 $q' \Rrightarrow_k q$ (defined in Fig. 23) for the zero or multiple-step executions from the abstract code specified by q' to the code specified by q. It also requires the number of $\blacklozenge$-tokens at level k to be decreased if $k \geq 1$. That is, the current thread should lose $\blacklozenge$-tokens when it performs a level-k action that may delay other threads. 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, the concrete steps and the level k) 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.

The (CSQ) rule uses $p' \overset{G}{\Longrightarrow} p$ and $q \overset{G}{\Longrightarrow} q'$. The definition of $p \overset{G}{\Longrightarrow} q$ is similar to $p \Rrightarrow_k q$, as defined in Fig. 23. In addition to executing the abstract code and decreasing the corresponding $\blacklozenge$-tokens, $p \overset{G}{\Longrightarrow} q$ also requires the overall transition to satisfy G.

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

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

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

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

$$
C \uplus C' \stackrel{\text{def}}{=} \begin{cases} C' & \text{if } C = \textbf{skip} \\ C & \text{if } C' = \textbf{skip} \end{cases}
\qquad
\begin{array}{l}
(\mathfrak{S},(u,w),C) \uplus (\mathfrak{S}',(u',w'),C') \stackrel{\text{def}}{=} \\
\qquad (\mathfrak{S} \uplus \mathfrak{S}', (u+u', w+w'), C \uplus C')
\end{array}
$$

(b) Semantics of full assertions p and q.

$$
\begin{array}{lll}
(\mathfrak{S},\mathfrak{S}') \models P \ltimes_{k'} Q & \text{iff} & (\mathfrak{S} \models P) \wedge (\mathfrak{S}' \models Q) \\
(\mathfrak{S},\mathfrak{S}') \models [P] & \text{iff} & (\mathfrak{S}' = \mathfrak{S}) \wedge (\mathfrak{S} \models P) \\
(\mathfrak{S},\mathfrak{S}') \models G_1 * G_2 & \text{iff} & \exists \mathfrak{S}_1, \mathfrak{S}_2, \mathfrak{S}_1', \mathfrak{S}_2'.\, \mathfrak{S} = \mathfrak{S}_1 \uplus \mathfrak{S}_2 \wedge \mathfrak{S}' = \mathfrak{S}_1' \uplus \mathfrak{S}_2' \\
& & \qquad \wedge ((\mathfrak{S}_1,\mathfrak{S}_1') \models G_1) \wedge ((\mathfrak{S}_2,\mathfrak{S}_2') \models G_2)
\end{array}
$$

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

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

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

Fig. 21. Semantics of assertions.

$$\mathcal{L}((\mathfrak{S},\mathfrak{S}'),P \ltimes_k Q) \stackrel{\text{def}}{=} \begin{cases} k & \text{if } (\mathfrak{S},\mathfrak{S}') \models P \ltimes_k Q \\ \mathsf{maxL} & \text{otherwise} \end{cases}$$

$$\mathcal{L}((\mathfrak{S},\mathfrak{S}'),[P]) \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } (\mathfrak{S},\mathfrak{S}') \models [P] \\ \mathsf{maxL} & \text{otherwise} \end{cases}$$

$$\mathcal{L}((\mathfrak{S},\mathfrak{S}'),\mathcal{D}) \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } (\mathfrak{S},\mathfrak{S}') \models \mathcal{D} \\ \mathsf{maxL} & \text{otherwise} \end{cases}$$

$$\mathcal{L}((\mathfrak{S},\mathfrak{S}'),R \wedge R') \stackrel{\text{def}}{=} \max(\mathcal{L}((\mathfrak{S},\mathfrak{S}'),R),\mathcal{L}((\mathfrak{S},\mathfrak{S}'),R'))$$

$$\mathcal{L}((\mathfrak{S},\mathfrak{S}'),R \vee R') \stackrel{\text{def}}{=} \min(\mathcal{L}((\mathfrak{S},\mathfrak{S}'),R),\mathcal{L}((\mathfrak{S},\mathfrak{S}'),R'))$$

$$\mathcal{L}((\mathfrak{S},\mathfrak{S}'),\lfloor R\rfloor_0) \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } \mathcal{L}((\mathfrak{S},\mathfrak{S}'),R) = 0 \\ \mathsf{maxL} & \text{otherwise} \end{cases}$$

$$\mathcal{L}((\mathfrak{S},\mathfrak{S}'),G_1 * G_2) \stackrel{\text{def}}{=} min\{k \mid \exists \mathfrak{S}_1,\mathfrak{S}_2,\mathfrak{S}_1',\mathfrak{S}_2'.\ (\mathfrak{S}=\mathfrak{S}_1 \uplus \mathfrak{S}_2) \wedge (\mathfrak{S}'=\mathfrak{S}_1' \uplus \mathfrak{S}_2') \wedge k = max(\mathcal{L}((\mathfrak{S}_1,\mathfrak{S}_1'),G_1),\mathcal{L}((\mathfrak{S}_2,\mathfrak{S}_2'),G_2))\}$$

$$(\mathfrak{S},\mathfrak{S}') \models \lfloor R\rfloor_0 \ \text{ iff } \ \mathcal{L}((\mathfrak{S},\mathfrak{S}'),R) = 0$$

$$(\mathfrak{S},\mathfrak{S}',k) \models R \ \text{ iff } \ \mathcal{L}((\mathfrak{S},\mathfrak{S}'),R) = k \text{ and } k < \mathsf{maxL}$$

$$R \Rrightarrow R' \quad \text{iff } \forall \mathfrak{S},\mathfrak{S}',k.\ ((\mathfrak{S},\mathfrak{S}',k) \models R) \Longrightarrow \exists k' \le k.\ (\mathfrak{S},\mathfrak{S}',k') \models R'$$

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

$$(n'_m,\ldots,n'_1) <_k (n_m,\ldots,n_1) \ \text{ iff } \ (\forall i > k.\ (n'_i = n_i)) \wedge (n'_k < n_k)$$

$$(n'_m,\ldots,n'_1) \approx_k (n_m,\ldots,n_1) \ \text{ iff } \ (\forall i \ge k.\ (n'_i = n_i))$$

$$u < u' \ \text{ iff } \ \exists k.\ u <_k u' \qquad\qquad u \le u' \ \text{ iff } \ u < u' \vee u = u'$$

$$(u,w) <_k (u',w') \ \text{ iff } \ (u <_k u') \vee (k = 0 \wedge u = u' \wedge w < w')$$

$$(u,w) \approx_k (u',w') \ \text{ iff } \ u \approx_k u' \wedge (k = 0 \Longrightarrow w = w')$$

Fig. 22. Levels of state transitions and tokens.

$$\begin{aligned} p \Rrightarrow_k q \ \text{ iff } \ & \forall \mathsf{t},\sigma,\Sigma,u,w,C,\Sigma_F. \\ & (((\sigma,\Sigma),(u,w),C) \models p) \wedge (\Sigma \perp \Sigma_F) \Longrightarrow \exists u',w',C',\Sigma'. \\ & ((C,\Sigma \uplus \Sigma_F) \longrightarrow_{\mathsf{t}}^* (C',\Sigma' \uplus \Sigma_F)) \wedge (((\sigma,\Sigma'),(u',w'),C') \models q) \\ & \wedge (u',w') <_k (u,w) \end{aligned}$$

$$\begin{aligned} p \overset{G}{\Longrightarrow} q \ \text{ iff } \ & \forall \mathsf{t},\sigma,\Sigma,u,w,C,\Sigma_F. \\ & (((\sigma,\Sigma),(u,w),C) \models p) \wedge (\Sigma \perp \Sigma_F) \Longrightarrow \exists k,u',w',C',\Sigma'. \\ & ((C,\Sigma \uplus \Sigma_F) \longrightarrow_{\mathsf{t}}^* (C',\Sigma' \uplus \Sigma_F)) \wedge (((\sigma,\Sigma),(\sigma,\Sigma'),k) \models G * \mathsf{True}) \\ & \wedge (((\sigma,\Sigma'),(u',w'),C') \models q) \wedge (u',w') <_k (u,w) \end{aligned}$$

$$\begin{aligned} \mathsf{Sta}(p,R) \ \text{ iff } \ & \forall \mathfrak{S},\mathfrak{S}',u,w,C,k. \\ & ((\mathfrak{S},(u,w),C) \models p) \wedge ((\mathfrak{S},\mathfrak{S}',k) \models R) \Longrightarrow \exists u',w'. \\ & ((\mathfrak{S}',(u',w'),C) \models p) \ \wedge \ ((u',w') \approx_k (u,w)) \end{aligned}$$

Fig. 23. Key auxiliary definitions for inference rules.

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

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

$$\frac{\begin{array}{c}\mathfrak{L}', \mathcal{D}', R, G, I \vdash \{p\}C\{q\} \quad \mathcal{D} = \mathcal{D}' \wedge \mathcal{D}'' \quad \mathfrak{L}' \leqslant \mathfrak{L} \\ J \Rightarrow I \quad p \Rightarrow J * \mathsf{true} \quad \mathsf{Sta}(J, R \vee G) \quad \mathsf{layGt}(\mathfrak{L}, \mathcal{D}'', J, \mathcal{D}')\end{array}}{\mathfrak{L}, \mathcal{D}, R, G, I \vdash \{p\}C\{q\}}\ (\textsc{live-frm})$$

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

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

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

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

$$\frac{\begin{array}{c}\mathfrak{L}, \mathcal{D}, R, G, I \vdash \{p\}C_1\{r\} \\ \mathfrak{L}, \mathcal{D}, R, G, I \vdash \{r\}C_2\{q\}\end{array}}{\mathfrak{L}, \mathcal{D}, R, G, I \vdash \{p\}C_1; C_2\{q\}}\ (\textsc{seq}) \qquad \frac{\begin{array}{c}p \Rightarrow (B = B) * I \quad \mathfrak{L}, \mathcal{D}, R, G, I \vdash \{p \wedge B\}C_1\{q\} \\ \mathfrak{L}, \mathcal{D}, R, G, I \vdash \{p \wedge \neg B\}C_2\{q\}\end{array}}{\mathfrak{L}, \mathcal{D}, R, G, I \vdash \{p\}\mathbf{if}\ (B)\ C_1\ \mathbf{else}\ C_2\{q\}}\ (\textsc{if})$$

$$\frac{\mathfrak{L}, \mathcal{D}, R, G, I \vdash \{p\}C\{q\}}{\mathfrak{L}, \mathcal{D}, R, G, I \vdash \{\lfloor p \rfloor_{\Diamond}\}C\{\lfloor q \rfloor_{\Diamond}\}}\ (\textsc{hide-}\Diamond) \qquad \frac{\mathfrak{L}, \mathcal{D}, R, G, I \vdash \{p\}C\{q\}}{\mathfrak{L}, \mathcal{D}, R, G, I \vdash \{p * r\}C\{q * r\}}\ (\textsc{frm})$$

$$\frac{\begin{array}{c}\vdash [p]C[q] \quad \mathsf{Sta}(r, R * \mathsf{Id}) \\ I \triangleright \{G, R\} \quad r \Rightarrow I * \mathsf{true}\end{array}}{\mathfrak{L}, \mathcal{D}, R, G, I \vdash \{p * r\}C\{q * r\}}\ (\textsc{prim}) \qquad \frac{\begin{array}{c}\mathfrak{L}, \mathcal{D}, R, G, I \vdash \{p\}C\{q\} \\ x \notin fv(\mathcal{D}, R, G)\end{array}}{\mathfrak{L}, \mathcal{D}, R, G, I \vdash \{\exists x.\ p\}C\{\exists x.\ q\}}\ (\textsc{ex})$$

$$\frac{\begin{array}{c}\mathfrak{L}, \mathcal{D}, R, G, I \vdash \{p_1\}C\{q_1\} \\ \mathfrak{L}, \mathcal{D}, R, G, I \vdash \{p_2\}C\{q_2\}\end{array}}{\mathfrak{L}, \mathcal{D}, R, G, I \vdash \{p_1 \wedge p_2\}C\{q_1 \wedge q_2\}}\ (\textsc{conj}) \qquad \frac{\begin{array}{c}\mathfrak{L}, \mathcal{D}, R, G, I \vdash \{p_1\}C\{q_1\} \\ \mathfrak{L}, \mathcal{D}, R, G, I \vdash \{p_2\}C\{q_2\}\end{array}}{\mathfrak{L}, \mathcal{D}, R, G, I \vdash \{p_1 \vee p_2\}C\{q_1 \vee q_2\}}\ (\textsc{disj})$$

Fig. 24. Inference rules.

B LOGIC SOUNDNESS PROOFS

B.1 Overview of the structures of soundness proofs

The top at Fig. 25 shows our final goals: verifying linearizability $\Pi \preceq_P^{\text{lin}} \Gamma$, and starvation-free$_P(\Pi)$ or deadlock-free$_P(\Pi)$. Inspired by earlier work [Filipović et al. 2010; Liang and Feng 2016; Liang et al. 2013], we reduce the goals (see ① and ② in Fig. 25) to verifying contextual refinements $\sqsubseteq$.

When using $\sqsubseteq$ to characterize deadlock-free objects Π (see ② in Fig. 25), their methods at the abstract side are no longer atomic. We need to use the "progress-aware" object specification $\text{wr}_1(\Gamma)$.

Our results ① and ② in Fig. 25 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. 25.) 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 [Liang and Feng 2013, 2016; Liang et al. 2014].

At the bottom ⑨ of Fig. 25, we define semantics for our logic judgment $\mathfrak{L}, \mathcal{D}, R, G, I \vdash \{P\}\Pi : \Gamma$. Note that the logic uses the atomic Γ as specification. The judgment semantics $\mathfrak{L}, \mathcal{D}, R, G \models \{P\}\Pi : \Gamma$ 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. 24. It can be directly (see ⑧) translated to $\Pi \precsim_{\text{wr}_1(P)} \text{wr}_1(\Gamma)$, where wr_1 will be defined later. If the rely/guarantee conditions R and G specify actions of level 0 only (see case (2) in Theorem 4.2), the judgment semantics can also be reduced to $\Pi \precsim_P \Gamma$ (see ⑦). Both $\Pi \precsim_{\text{wr}_1(P)} \text{wr}_1(\Gamma)$ and $\Pi \precsim_P \Gamma$ are instances of the more general simulation $\Pi \precsim_{P'} \Pi'$ (see ⑤ and ⑥).

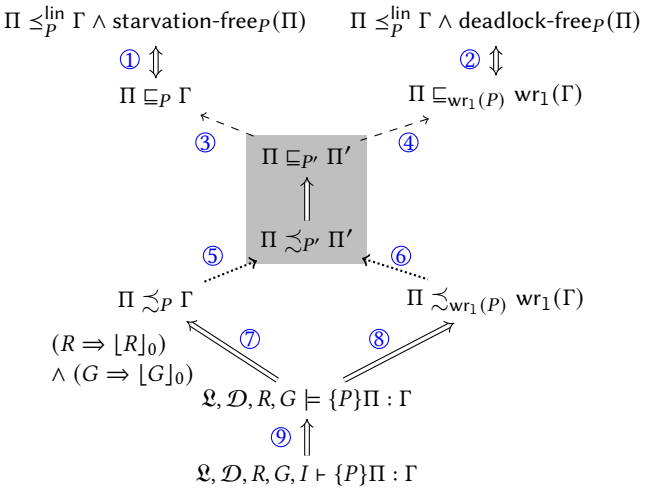

Fig. 25. A unified framework for logic soundness proofs.

We give the detailed proofs of Theorem 4.2 following the unified proof framework in Fig. 25.

- Appendix B.2 defines the judgment semantics $\mathfrak{L}, \mathcal{D}, R, G \models \{P\}\Pi : \Gamma$.
- Appendix B.3 shows the proofs of ⑨ of Fig. 25, i.e., the logic rules are sound with respect to the judgment semantics.
- Appendix B.4 defines the "common simulation" $\Pi \precsim_{P'} \Pi'$. Appendix B.4.1 shows the proofs of ⑦ and ⑧ of Fig. 25, i.e., the judgment semantics $\mathfrak{L}, \mathcal{D}, R, G \models \{P\}\Pi : \Gamma$ implies instantiations of the common simulation.

- Appendix B.5 shows the core proofs for the gray box at the center of Fig. 25, i.e., the common simulation implies $\Pi \sqsubseteq_{P'} \Pi'$, the contextual refinement under fair scheduling.
- Appendix B.6 shows the proofs of ① and ② of Fig. 25, i.e., deadlock-freedom/starvation-freedom and linearizability can be characterized by the contextual refinements.

B.2 Judgment Semantics

The judgment semantics $\mathfrak{L}, \mathcal{D}, R, G \models \{P\}\Pi : \Gamma$ 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 pairs of the environment thread and its enabled base definite action that t is waiting for. It shrinks when an environment thread t′ finishes the base definite action $\mathcal{B}_{\mathsf{t}'}$. The metric $\mathbb{M}$ corresponds to the number of white tokens $\Diamond$, 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 B.1. $\mathfrak{L}, \mathcal{D}, R, G \models \{P\}\Pi : \Gamma$ iff, for any $f \in dom(\Pi)$, 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 $\xi \in \mathscr{P}(ThrdID \times BaseDAct)$ such that

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

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

(1) Suppose $\sigma = (s, h)$, and $\delta = \{\mathcal{B} \mid (\mathcal{B} \leqslant \mathcal{D}) \wedge ((\sigma, \Sigma) \models \mathsf{Enabled}(\mathcal{B}_\mathsf{t}) * \mathbf{true})\}$, then
  (a) $\mathtt{tids}(\xi) \subseteq s(\mathtt{TIDS})$ and
  (b) for any $(\mathsf{t}', \mathcal{B}') \in \xi$, $\mathsf{t}' \neq \mathsf{t}$ and there exists $\mathfrak{S}'$ such that
    (b1) there exists $\mathfrak{S}''$, $(\sigma, \Sigma) = \mathfrak{S}' \uplus \mathfrak{S}''$ and $\mathfrak{S}' \models \mathsf{Enabled}(\mathcal{B}'_{\mathsf{t}'})$
    (b2) $\mathfrak{L}(\mathfrak{S}', \mathcal{B}')$ is defined and
    (b3) for any $\mathcal{B} \in \delta$, $\mathfrak{L}(\mathfrak{S}', \mathcal{B}) > \mathfrak{L}(\mathfrak{S}', \mathcal{B}')$

(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) $\mathfrak{L}, \mathcal{D}, R, G \models_\mathsf{t} (C', \sigma') \preceq (\mathbb{C}', \Sigma') \diamond (u', \mathbb{M}', M') \Downarrow_{\xi'} Q$, 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) either $M' < M$,
    or $M' = M$ and $\xi \neq \emptyset$ and $\xi \subseteq \xi'$

(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) $\mathfrak{L}, \mathcal{D}, R, G \models_\mathsf{t} (C, \sigma') \preceq (\mathbb{C}, \Sigma') \diamond (u', \mathbb{M}', M') \Downarrow_{\xi'} Q$, and
  (b) $\xi_d = \{(\mathsf{t}', \mathcal{B}) \mid ((\mathsf{t}', \mathcal{B}) \in \xi) \wedge (((\sigma, \Sigma), (\sigma', \Sigma')) \models \langle \mathcal{B}_{\mathsf{t}'} \rangle * \mathsf{Id})\}$ 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 $k = 0$, then either $M' < M$ or $M' = M$ and $\xi_d = \emptyset$.

Definition B.2. $\mathfrak{L}, \mathcal{D}, R, G \models \{p\}C\{q\}$ 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

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

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

(1) Suppose $\sigma = (s, h)$, and $\delta = \{\mathcal{B} \mid (\mathcal{B} \leqslant \mathcal{D}) \wedge ((\sigma, \Sigma) \models \mathsf{Enabled}(\mathcal{B}_{\mathsf{t}}) * \mathbf{true})\}$, then
  (a) $\mathtt{tids}(\xi) \subseteq s(\mathtt{TIDS})$ and
  (b) for any $(\mathsf{t}', \mathcal{B}') \in \xi$, $\mathsf{t}' \neq \mathsf{t}$ and there exists $\mathfrak{S}'$ such that
    (b1) there exists $\mathfrak{S}''$, $(\sigma, \Sigma) = \mathfrak{S}' \uplus \mathfrak{S}''$ and $\mathfrak{S}' \models \mathsf{Enabled}(\mathcal{B}'_{\mathsf{t}'})$
    (b2) $\mathfrak{L}(\mathfrak{S}', \mathcal{B}')$ is defined and
    (b3) for any $\mathcal{B} \in \delta$, $\mathfrak{L}(\mathfrak{S}', \mathcal{B}) > \mathfrak{L}(\mathfrak{S}', \mathcal{B}')$
  (c) If $\xi \neq \emptyset$, then $|\mathbb{ws}| > 1$.
(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) $\mathbb{ws} = ((0,0), 0)$ and $ws = ((0,0), 0)$ 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 $\Sigma \perp \Sigma_F$, 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) $((\sigma, \Sigma'), (u, w), \mathbf{skip}) \models q_{\mathsf{t}}$ 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) $\mathfrak{L}, \mathcal{D}, R, G \models_{\mathsf{t}} (C', \sigma') \preceq (\mathbb{C}', \Sigma') \diamond (u', \mathbb{ws}', ws', w', \mathcal{H}) \Downarrow_{\xi'} q$, 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) $ws' <_{\mathcal{H}} ws$,
    or $ws' = ws$ and $\xi \neq \emptyset$ and $\xi \subseteq \xi'$
(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) $\mathfrak{L}, \mathcal{D}, R, G \models_{\mathsf{t}} (C, \sigma') \preceq (\mathbb{C}, \Sigma') \diamond (u', \mathbb{ws}', ws', w', \mathcal{H}) \Downarrow_{\xi'} q$, and
  (b) $\xi_d = \{(\mathsf{t}', \mathcal{B}) \mid ((\mathsf{t}', \mathcal{B}) \in \xi) \wedge (((\sigma, \Sigma), (\sigma', \Sigma')) \models \langle \mathcal{B}_{\mathsf{t}'} \rangle * \mathsf{Id})\}$ 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 $k = 0$, then either $ws' <_{\mathcal{H}} ws$, or $ws' = ws$ and $\xi_d = \emptyset$.

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 B.3 (SL judgment semantics, total correctness). $\models [p]C[q]$ iff, for all σ, Σ, u, w and $\mathbb{C}$, for any t, if $((\sigma,\Sigma),(u,w),\mathbb{C}) \models p_t$, the following are true:

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

Lemma B.4. *If* $\vdash [p]C[q]$, *then* $\models [p]C[q]$.

Definition B.5 (Locality).
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 $(C,\sigma_1) \not\longrightarrow_t^* \mathbf{abort}$, then $(C,\sigma) \not\longrightarrow_t^* \mathbf{abort}$.
(2) (Termination monotonicity) If $(C,\sigma_1) \not\longrightarrow_t^* \mathbf{abort}$ and $(C,\sigma_1) \not\longrightarrow_t^\omega \cdot$, then $(C,\sigma) \not\longrightarrow_t^\omega \cdot$.
(3) (Frame property) For any n and σ', if $(C,\sigma_1) \not\longrightarrow_t^* \mathbf{abort}$ and $(C,\sigma) \longrightarrow_t^n (C',\sigma')$, then there exists σ_1' such that $\sigma' = \sigma_1' \uplus \sigma_2$ and $(C,\sigma_1) \longrightarrow_t^n (C',\sigma_1')$.

B.2.1 Instantiating Metrics and Well-Founded Orders. The judgment semantics in Definition B.2 can be viewed as an instantiation of the simulation in Definition B.1. The key is to instantiate the metrics $\mathbb{M}$ and M in $\mathfrak{L},\mathcal{D},R,G \models_t (C,\sigma) \preceq (\mathbb{C},\Sigma) \diamond (u,\mathbb{M},M) \Downarrow_\xi Q$. Below we take the instantiation for M as an example.

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

For each single thread, its metric ws is usually a list of (wn,n) pairs, where wn 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 [Liang et al. 2014]. 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 (wn,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.

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

$$ws' \prec_{\mathcal{H}} ws \quad \text{iff} \quad (ws' \prec ws) \wedge (|ws'| \le \mathcal{H}) \wedge (|ws| \le \mathcal{H})$$

$$ws' \preccurlyeq_{\mathcal{H}} ws \quad \text{iff} \quad (ws' \prec_{\mathcal{H}} ws) \vee (ws' = ws)$$

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

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

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

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

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

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

$$\begin{array}{lll} (wn', n') < (wn, n) & \text{iff} & (wn' < wn) \vee (wn' = wn \wedge n' < n) \\ (wn', n') = (wn, n) & \text{iff} & (wn' = wn) \wedge (n' = n) \\ (wn', n') \leq (wn, n) & \text{iff} & (wn', n') < (wn, n) \vee (wn', n') = (wn, n) \end{array}$$

Lemma B.6 (Well-foundedness). *The relations $M' < M$ and $ws' <_{\mathcal{H}} ws$ defined above are both well-founded relations.*

The judgment semantics in Definition B.2 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.

$$\begin{array}{lll} ws' <_{\mathcal{H}} ws & \text{iff} & (ws' \ll ws) \wedge (|ws'| \leq \mathcal{H}) \wedge (|ws| \leq \mathcal{H}) \\ ws' \leq_{\mathcal{H}} ws & \text{iff} & (ws' <_{\mathcal{H}} ws) \vee (ws' = ws) \end{array}$$

$$\frac{(wn', n') < (wn, n)}{(wn', n') \ll (wn, n)} \qquad \frac{(wn', n') = (wn, n) \qquad ws'_1 \ll ws_1}{(wn', n') :: ws'_1 \ll (wn, n) :: ws_1}$$

$$\frac{(wn', n') < (wn, n)}{(wn', n') :: ws'_1 \ll (wn, n)} \qquad \frac{(wn', n') = (wn, n)}{(wn', n') \ll (wn, n) :: ws_1}$$

$$\frac{(wn'_0, n'_0) = (wn_0, n_0) \qquad (wn', n') < (wn, n)}{(wn'_0, n'_0) :: (wn', n') :: ws'_1 \ll (wn_0, n_0) :: (wn, n) :: ws_1}$$

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.

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

Initial code size. In Definition B.2, the judgment semantics initially takes the static code size $|C|$ defined below as the second dimension of ws and $\mathbb{ws}$.

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

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

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.3 Soundness of the Inference Rules

In this section, we prove Lemma B.7 by induction over the derivation.

Lemma B.7 (⑨ in Fig. 25). *If* $\mathfrak{L}, \mathcal{D}, R, G, I \vdash \{P\}\Pi : \Gamma$, *then* $\mathfrak{L}, \mathcal{D}, R, G \models \{P\}\Pi : \Gamma$.

Proof. By induction over derivation. By Lemma B.8, 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
> $\mathfrak{L}, \mathcal{D}, R, G \models \{P * \mathsf{own}(x) * \mathsf{own}(y) \wedge (x = y) \wedge \mathsf{arem}(\mathbb{C}) \wedge \blacklozenge(E_k, \ldots, E_1)\}\ C\ \{P * \mathsf{own}(x) * \mathsf{own}(y) \wedge \mathsf{arem}(\mathbf{skip})\}$,
> then $\mathfrak{L}, \mathcal{D}, R, G \models \{P\}\Pi : \Gamma$. (B.1)

By co-induction. □

Lemma B.8. *If*

(1) $\mathfrak{L}, \mathcal{D}, R, G, I \vdash \{p\}C\{q\}$;
(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 $\mathfrak{L}, \mathcal{D}, R, G \models \{p\}C\{q\}$.

In the following subsections, we prove Lemma B.8 by induction over the derivation.

B.3.1 *The* WHL *Rule.*

Lemma B.9 (WHL-Sound). *If*

(1) $p \wedge B \Rightarrow p'$; $p \wedge B \wedge Q * \mathsf{true} \Rightarrow p' * (\Diamond \wedge \mathsf{emp})$;
(2) $\mathfrak{L}, \mathcal{D}, R, G \models \{p'\}C\{p\}$;
(3) $p \Rightarrow (B = B) * J$; $J \Rightarrow I$; $\mathsf{Sta}(J, R \vee G)$; $Q \Rightarrow I$;
(4) $J \Rightarrow (R, G : \mathcal{D}' \xrightarrow{f} Q)$; $\mathcal{D}' \leqslant \mathcal{D}$; $selD(\mathfrak{L}, \mathcal{D}, J, \mathcal{D}')$
(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 $\mathfrak{L}, \mathcal{D}, R, G \models \{p\}\mathbf{while}\ (B)\{C\}\{p \wedge \neg B\}$.

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

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

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

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

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

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

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) If $\delta = \{\mathcal{B} \mid (\mathcal{B} \leqslant \mathcal{D}) \wedge ((\sigma, \Sigma) \models \mathsf{Enabled}(\mathcal{B}_{\mathsf{t}}) * \mathbf{true})\}$, then for any $(\mathsf{t}', \mathcal{B}') \in \xi$, ...
Proof: It is vacantly true.

(2) If **while** $(B)\{C\} = \textbf{skip}$, then ....
Proof: It is vacantly true.
(3) If **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) $\mathfrak{L}, \mathcal{D}, R, G \models_{\mathsf{t}} (C', \sigma') \preceq (\mathbb{C}', \Sigma') \diamond (u', \mathbb{ws}', ws', w', \mathcal{H}) \Downarrow_{\xi'} p \wedge \neg B$, 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) either $ws' <_{\mathcal{H}} ws$,
  or $ws' = ws$ and $\xi \neq \emptyset$ and $\xi \subseteq \xi'$.

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

(I) If $[\![B]\!]_s = \textbf{true}$, 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 $\mathfrak{L}, \mathcal{D}, R, G \models \{p'\}C\{p\}$, let $\mathbb{ws}_1 = ((0,0), |C|)$ and $ws_1 = ((0,0), |C|)$, we get:

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

Let

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

Since $J \Rightarrow (R, G : \mathcal{D}' \xrightarrow{f} Q)$, 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 B.10, let
  $$\mathbb{ws}' = ((0,0),0) :: ((k_s, w), 0)\ ,\ ws' = ((0,0),0) :: ((k_s, w), 0)\ .$$
  we know
  $$\mathfrak{L}, \mathcal{D}, R, G \models_{\mathsf{t}} (C; \mathsf{while}\ (B)\{C\}, \sigma) \preceq (\mathbb{C}, \Sigma) \diamond (u, \mathbb{ws}', ws', w, \mathcal{H}) \Downarrow_{\xi_0} p \wedge \neg B\ .$$
  Also we know $\mathbb{ws}' <_{\mathcal{H}} \mathbb{ws}$ and $ws' <_{\mathcal{H}} ws$.
- $(\sigma, \Sigma) \models Q_{\mathsf{t}} * \mathsf{true}$.
  By Lemma B.10, let
  $$\mathbb{ws}' = ((0,0),0) :: ((k_s, w), |C|+1)\ ,\ ws' = ((0,0),0) :: ((k_s, w), |C|+1)\ ,$$
  we know
  $$\mathfrak{L}, \mathcal{D}, R, G \models_{\mathsf{t}} (C; \mathsf{while}\ (B)\{C\}, \sigma) \preceq (\mathbb{C}, \Sigma) \diamond (u, \mathbb{ws}', ws', w, \mathcal{H}) \Downarrow_{\xi_0} p \wedge \neg B\ .$$
  Also, $\mathbb{ws}' <_{\mathcal{H}} \mathbb{ws}$ and $ws' <_{\mathcal{H}} ws$ holds.

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

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

we know

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

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) $\mathfrak{L},\mathcal{D},R,G \models_{\mathsf{t}} (\textbf{while}\ (B)\{C\},\sigma') \preceq (\mathbb{C},\Sigma') \diamond (u',\mathbb{ws}',ws',w',\mathcal{H}) \Downarrow_{\xi'} p \wedge \neg B$, and
   (b) $\xi_d = \{(\mathsf{t}',\mathcal{B}) \mid ((\mathsf{t}',\mathcal{B}) \in \xi) \wedge (((\sigma,\Sigma),(\sigma',\Sigma')) \models \langle \mathcal{B}_{\mathsf{t}'} \rangle * \mathsf{Id})\}$ and $(k = 0 \implies \xi \backslash \xi_d \subseteq \xi')$ and $u' \approx_k u$, and
   (c) if $k = 0$, then either $\mathbb{ws}' <_{\mathcal{H}} \mathbb{ws}$ and $w' = w$, or $\mathbb{ws}' = \mathbb{ws}$ and $w' = w$ and $\xi_d = \emptyset$; and
   (d) if $k = 0$, then either $ws' <_{\mathcal{H}} ws$, or $ws' = ws$ and $\xi_d = \emptyset$.

   Proof: Since $\mathsf{Sta}(p, R * \mathsf{Id})$, we know there exist u' and w' such that

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

   By the co-induction hypothesis, we get

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

   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 B.10. *If*

(1) $p \wedge B \Rightarrow p'$; $p \wedge B \wedge Q * \mathsf{true} \Rightarrow p' * (\Diamond \wedge \mathsf{emp})$;
(2) $\mathfrak{L},\mathcal{D},R,G \models \{p'\}C\{p\}$;
(3) $p \Rightarrow (B = B) * J$; $J \Rightarrow I$; $\mathsf{Sta}(J, R \vee G)$; $Q \Rightarrow I$;
(4) $J \Rightarrow (R, G\colon \mathcal{D}' \xrightarrow{f} Q)$; $\mathcal{D}' \leqslant \mathcal{D}$; $selD(\mathfrak{L},\mathcal{D},J,\mathcal{D}')$
(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) $\mathfrak{L},\mathcal{D},R,G \models_{\mathsf{t}} (C_1,\sigma) \preceq (\mathbb{C},\Sigma) \diamond (u,\mathbb{ws}_1,ws_1,w_1,\mathcal{H}) \Downarrow_{\xi_1} p$;
(8) $(\sigma,\Sigma) \models J * \mathsf{true}$; $\mathsf{height}(C) = \mathcal{H}$; $\mathsf{head}(\mathbb{ws}_1) = ((n_1,n_2),n_3)$; $\mathsf{head}(ws_1) = ((\mathbb{n}_1,\mathbb{n}_2),\mathbb{n}_3)$; $f(\sigma,\Sigma,i) = k_s$; $w_1 \leq w$;
(9) $\xi_0 = \{(\mathsf{t}',\mathcal{B}) \mid (\mathsf{t}' \neq \mathsf{t}) \wedge (\mathcal{B} \leqslant \mathcal{D}) \wedge ((\sigma,\Sigma) \models \mathsf{Enabled}(\mathcal{B}_{\mathsf{t}'}) * \mathsf{true})\}$; $\xi = \xi_0 \cup \xi_1$;
(10) *one of the following holds:*
- $\xi_0 \neq \emptyset$; $\mathbb{ws} = ((0,0),0) :: \mathsf{inchead}(\mathbb{ws}_1,((k_s,w_1),-n_3))$; $ws = ((0,0),0) :: \mathsf{inchead}(ws_1,((k_s,w_1),-\mathbb{n}_3))$
- $(\sigma,\Sigma) \models Q_{\mathsf{t}} * \mathsf{true}$; $\mathbb{ws} = ((0,0),0) :: \mathsf{inchead}(\mathbb{ws}_1,((k_s,w_1),1))$; $ws = ((0,0),0) :: \mathsf{inchead}(ws_1,((k_s,w_1),1))$

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

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

(1) if $\delta = \{\mathcal{B} \mid (\mathcal{B} \leqslant \mathcal{D}) \wedge ((\sigma,\Sigma) \models \mathsf{Enabled}(\mathcal{B}_{\mathsf{t}}) * \textbf{true})\}$, then for any $(\mathsf{t}',\mathcal{B}') \in \xi$, $\mathsf{t}' \neq \mathsf{t}$ and there exists $\mathfrak{S}'$ such that
   (1) there exists $\mathfrak{S}''$, $(\sigma,\Sigma) = \mathfrak{S}' \uplus \mathfrak{S}''$ and $\mathfrak{S}' \models \mathsf{Enabled}(\mathcal{B}'_{\mathsf{t}'})$
   (2) $\mathfrak{L}(\mathfrak{S}',\mathcal{B}')$ is defined and
   (3) for any $\mathcal{B} \in \delta$, $\mathfrak{L}(\mathfrak{S}',\mathcal{B}) > \mathfrak{L}(\mathfrak{S}',\mathcal{B}')$

   Proof: From

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

we know for any $(\mathsf{t}', \mathcal{B}) \in \xi_1$, we have $\mathsf{t}' \neq \mathsf{t}$ and $\mathcal{B} \leqslant \mathcal{D}$ and there exists $\mathfrak{S}'$ such that (1) (2) (3) hold.
For any $(\mathsf{t}', \mathcal{B}) \in \xi_0$, we know $\mathsf{t}' \neq \mathsf{t}$ and $\mathcal{B} \leqslant \mathcal{D}'$, since $\mathcal{D}' \leqslant \mathcal{D}$, we know $\mathcal{B} \leqslant \mathcal{D}$.
From $(\sigma, \Sigma) \models \mathsf{Enabled}(\mathcal{B}_{\mathsf{t}'}) * \mathsf{true}$ and $\mathsf{Enabled}(\mathcal{D}) \Rightarrow I$, we can prove there exists $\mathfrak{S}'$ such that $\mathfrak{S}' \models I$ and (1) holds.
Since $(\sigma, \Sigma) \models J * \mathsf{true}$ and $J \Rightarrow I$, there exists $\mathfrak{S}''$ such that $\mathfrak{S}'' \models J \wedge I$, from $\mathsf{Precise}(I)$, we can prove $\mathfrak{S}' = \mathfrak{S}''$, by $\mathsf{selD}(\mathfrak{L}, \mathcal{D}, J, \mathcal{D}')$, we know (2) (3) hold. 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

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

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) $\mathfrak{L}, \mathcal{D}, R, G \models_{\mathsf{t}} (C', \sigma') \preceq (\mathbb{C}', \Sigma') \diamond (u', \mathbb{ws}', w', \mathcal{H} + 1) \Downarrow_{\xi'} p \wedge \neg B$, 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) either $ws' <_{\mathcal{H}} ws$,
   or $ws' = ws$ and $\xi \neq \emptyset$ and $\xi \subseteq \xi'$.

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 $\mathfrak{L}, \mathcal{D}, R, G \models_{\mathsf{t}} (C_1, \sigma) \preceq (\mathbb{C}, \Sigma) \diamond (u, \mathbb{ws}_1, ws_1, w_1, \mathcal{H}) \Downarrow_{\xi_1} p$, 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) $\mathfrak{L}, \mathcal{D}, R, G \models_{\mathsf{t}} (C_1', \sigma') \preceq (\mathbb{C}', \Sigma') \diamond (u', \mathbb{ws}_1', ws_1', w_1', \mathcal{H}) \Downarrow_{\xi_1'} p$, 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) either $ws_1' <_{\mathcal{H}} ws_1$,
   or $ws_1' = ws_1$ and $\xi \neq \emptyset$ and $\xi \subseteq \xi'$.

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

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

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

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

Suppose $\mathsf{head}(\mathbb{ws}_1') = ((n_1', n_2'), n_3')$; $\mathsf{head}(ws_1') = ((\mathbb{n}_1', \mathbb{n}_2'), \mathbb{n}_3')$;

- If $\xi_0' \neq \emptyset$, let
$$\mathbb{ws}' = ((0,0),0) :: \mathsf{inchead}(\mathbb{ws}_1', ((k_s', w_1), -n_3')) \;\text{ and}$$
$$ws = ((0,0),0) :: \mathsf{inchead}(ws_1', ((k_s', w_1), -\mathbb{n}_3)) \;\; \xi' = \xi_0' \cup \xi_1' \,.$$
Then by the co-induction hypothesis, we know
$$\mathfrak{L}, \mathcal{D}, R, G \models_{\mathsf{t}} (C_1'; \mathsf{while}\ (B)\{C\}, \sigma') \preceq (\mathbb{C}', \Sigma') \diamond (u', \mathbb{ws}', ws', w', \mathcal{H}+1) \Downarrow_{\xi'} p \wedge \neg B \,.$$
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 $\xi \subseteq \xi'$.
  The reasoning for (f) is the same as (e).
- 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
$$\mathfrak{L}, \mathcal{D}, R, G \models_{\mathsf{t}} (C_1'; \mathsf{while}\ (B)\{C\}, \sigma') \preceq (\mathbb{C}', \Sigma') \diamond (u', \mathbb{ws}', ws', w', \mathcal{H}+1) \Downarrow_{\xi'} p \wedge \neg B \,.$$
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 $\xi \subseteq \xi'$.
The reasoning for (f) is the same as (e).

(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 $\mathfrak{L}, \mathcal{D}, R, G \models_{\mathsf{t}} (C_1, \sigma) \preceq (\mathbb{C}, \Sigma) \diamond (u, \mathbb{ws}_1, ws_1, w_1, \mathcal{H}) \Downarrow_{\xi_1} p$, 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 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
$$\xi_0' = \{(\mathsf{t}', \mathcal{B}) \mid (\mathsf{t}' \neq \mathsf{t}) \wedge (\mathcal{B} \leqslant \mathcal{D}') \wedge ((\sigma, \Sigma') \models \mathsf{Enabled}(\mathcal{B}_{\mathsf{t}'}) * \mathsf{true})\} \,.$$

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

$$ws' = ((0,0),0) :: \mathsf{inchead}(ws_1, ((k_s', w_1), 0)) \;;$$
$$\mathbb{ws}' = ((0,0),0) :: \mathsf{inchead}(\mathbb{ws}_1, ((k_s', w_1), 0)) \;;\; \xi' = \xi_0'$$

- $\xi_0 \neq \emptyset$, then $\xi \neq \emptyset$ and $\xi \subseteq \xi'$, and we know $ws' \leq_{\mathcal{H}+1} ws$ and $\mathbb{ws}' \leq_{\mathcal{H}+1} \mathbb{ws}$
- $(\sigma, \Sigma') \models Q_{\mathsf{t}} * \mathsf{true}$, then we know $ws' <_{\mathcal{H}+1} ws$ and $\mathbb{ws}' <_{\mathcal{H}+1} \mathbb{ws}$

Below we prove:

$$\mathfrak{L}, \mathcal{D}, R, G \models_{\mathsf{t}} (\mathsf{while}\ (B)\{C\}, \sigma) \preceq (\mathbb{C}', \Sigma') \diamond (u, \mathbb{ws}', ws', w, \mathcal{H}+1) \Downarrow_{\xi_0'} p \wedge \neg B \,. \tag{B.2}$$

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

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

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

(*1*) if $\delta = \{\mathcal{B} \mid (\mathcal{B} \leqslant \mathcal{D}) \wedge ((\sigma, \Sigma') \models \mathsf{Enabled}(\mathcal{B}_{\mathsf{t}}) * \mathbf{true})\}$, then for any $(\mathsf{t}', \mathcal{B}') \in \xi'$, $\mathsf{t}' \neq \mathsf{t}$ and there exists $\mathfrak{S}'$ such that
  (1) there exists $\mathfrak{S}''$, $(\sigma, \Sigma') = \mathfrak{S}' \uplus \mathfrak{S}''$ and $\mathfrak{S}' \models \mathsf{Enabled}(\mathcal{B}'_{\mathsf{t}'})$
  (2) $\mathfrak{L}(\mathfrak{S}', \mathcal{B}')$ is defined and
  (3) for any $\mathcal{B} \in \delta$, $\mathfrak{L}(\mathfrak{S}', \mathcal{B}) > \mathfrak{L}(\mathfrak{S}', \mathcal{B}')$

  Proof: For any $(\mathsf{t}', \mathcal{B}') \in \xi'$, we have $\mathsf{t}' \neq \mathsf{t}$ and $\mathcal{B}' \leqslant \mathcal{D}'$ and $(\sigma, \Sigma') \models \mathsf{Enabled}(\mathcal{B}'_{\mathsf{t}'}) * \mathsf{true}$, from $\mathcal{D}' \leqslant \mathcal{D}$, we have $\mathcal{B} \leqslant \mathcal{D}$. we can prove there exists $\mathfrak{S}'$ such that $\mathfrak{S}' \models I$ and (1) holds.
  Since $(\sigma, \Sigma') \models J * \mathsf{true}$ and $J \Rightarrow I$, there exists $\mathfrak{S}''$ such that $\mathfrak{S}'' \models J \wedge I$, from $\mathsf{Precise}(I)$, we can prove $\mathfrak{S}' = \mathfrak{S}''$, by $\mathsf{selD}(\mathfrak{L}, \mathcal{D}, J, \mathcal{D}')$, we know (2) (3) hold.

(*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) $\mathfrak{L}, \mathcal{D}, R, G \models_{\mathsf{t}} (C', \sigma') \preceq (\mathbb{C}'', \Sigma'') \diamond (u'', \mathbb{ws}'', ws'', w'', \mathcal{H}+1) \Downarrow_{\xi'} p \wedge \neg B$, 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) either $ws'' <_{\mathcal{H}} ws'$,
  or $ws'' = ws'$ and $\xi' \neq \emptyset$ and $\xi' \subseteq \xi''$.

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

  - If $[\![B]\!]_s = \mathbf{true}$, 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 $\xi_0' \neq \emptyset$,
      From $p \wedge B \Rightarrow p'$, we know
      $$((\sigma, \Sigma'), (u, w_1), \mathbb{C}') \models p_{\mathsf{t}}' \,.$$
      From $\mathfrak{L}, \mathcal{D}, R, G \models \{p'\}C\{p\}$ and $\mathsf{height}(C) = \mathcal{H}$, let $\mathbb{ws}_1' = ((0,0), |C|)$ and $ws_1' = ((0,0), |C|)$, we get:

$$\mathfrak{L}, \mathcal{D}, R, G \models_{\mathsf{t}} (C, \sigma) \preceq_i (\mathbb{C}', \Sigma') \diamond (u, \mathbb{ws}'_1, ws'_1, w_1, \mathcal{H}) \Downarrow_{\emptyset} p \,.$$

Let

$$\mathbb{ws}'' = ((0,0),0) :: ((k'_s, w_1), 0) \;\text{ and }\; ws'' = ((0,0),0) :: ((k'_s, w_1), 0) \,.$$

By the co-induction hypothesis, we know

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

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 $\mathfrak{L}, \mathcal{D}, R, G \models \{p'\}C\{p\}$ and $\mathsf{height}(C) = \mathcal{H}$, let $\mathbb{ws}'_1 = ((0,0), |C|)$, we get:
$$\mathfrak{L}, \mathcal{D}, R, G \models_{\mathsf{t}} (C, \sigma) \preceq_i (\mathbb{C}', \Sigma') \diamond (u, \mathbb{ws}'_1, w'_1, \mathcal{H}) \Downarrow_{\emptyset} p \,.$$
Let
$$\mathbb{ws}'' = ((0,0),0) :: ((k'_s, w'_1), |C|+1) \,.$$
By the co-induction hypothesis, we know
$$\mathfrak{L}, \mathcal{D}, R, G \models_{\mathsf{t}} (C; \mathsf{while}\ (B)\{C\}, \sigma) \preceq_i (\mathbb{C}', \Sigma') \diamond (u, \mathbb{ws}'', w, \mathcal{H}+1) \Downarrow_{\xi'_0} p \wedge \neg B \,.$$
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 $[\![B]\!]_s = \mathbf{false}$, we know $(\mathsf{while}\ (B)\{C\}, \sigma \uplus \sigma_F) \longrightarrow_{\mathsf{t}} (\mathbf{skip}, \sigma \uplus \sigma_F)$. By (SKIP) rule, let
$$\mathbb{ws}'' = ((0,0),0) \;\text{ and }\; ws'' = ((0,0),0) \,,$$
we know
$$\mathfrak{L}, \mathcal{D}, R, G \models_{\mathsf{t}} (\mathbf{skip}, \sigma) \preceq (\mathbb{C}', \Sigma') \diamond (u, \mathbb{ws}'', ws'', w, \mathcal{H}+1) \Downarrow_{\emptyset} p \wedge \neg B \,.$$
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) $\mathfrak{L}, \mathcal{D}, R, G \models_{\mathsf{t}} (\mathsf{while}\ (B)\{C\}, \sigma') \preceq (\mathbb{C}', \Sigma'') \diamond (u', \mathbb{ws}'', ws'', w'', \mathcal{H}+1) \Downarrow_{\xi'} p \wedge \neg B$, 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 $k = 0$, then either $ws'' <_{\mathcal{H}} ws'$, or $ws'' = ws'$ and $\xi_d = \emptyset$.

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
$\xi_0'' = \{(\mathsf{t}'', \mathcal{B}') \mid (\mathsf{t}'' \neq \mathsf{t}) \wedge (\mathcal{B}' \leqslant \mathcal{D}') \wedge ((\sigma', \Sigma'') \models \mathsf{Enabled}(\mathcal{B}'_{\mathsf{t}''}) * \mathsf{true})\}$ and
$\xi_d = \{(\mathsf{t}', \mathcal{B}) \mid ((\mathsf{t}', \mathcal{B}) \in \xi_0') \wedge (((\sigma, \Sigma'), (\sigma', \Sigma'')) \models \langle \mathcal{B}_{\mathsf{t}'} \rangle * \mathsf{Id})\}$.
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
$\mathfrak{L}, \mathcal{D}, R, G \models_{\mathsf{t}} (\mathsf{while}\ (B)\{C\}, \sigma') \preceq (\mathbb{C}, \Sigma'') \diamond (u, \mathbb{ws}'', ws', w'', \mathcal{H} + 1) \Downarrow_{\xi_0''} p \wedge \neg B$
.
Also we know: if $k = 0$, then $\mathbb{ws}'' \preceq_{\mathcal{H}+1} \mathbb{ws}'$ and $w'' = w$.
If $k = 0$ and $\xi_d \neq \emptyset$, then there exists $(\mathsf{t}', \mathcal{B})$ such that $(\mathsf{t}', \mathcal{B}) \in \xi_0'$ and $(((\sigma, \Sigma'), (\sigma', \Sigma'')) \models \langle \mathcal{B}_{\mathsf{t}'} \rangle * \mathsf{Id}$. Since $\mathcal{B} \leqslant \mathcal{D}$, 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
$$k_s'' < k_s' .$$
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) $\mathfrak{L}, \mathcal{D}, R, G \models_{\mathsf{t}} (C_1; \mathsf{while}\ (B)\{C\}, \sigma') \preceq (\mathbb{C}, \Sigma') \diamond (u', \mathbb{ws}', ws', w', \mathcal{H} + 1) \Downarrow_{\xi'} p \wedge \neg B$, and
- (b) $\xi_d = \{(\mathsf{t}', \mathcal{B}) \mid ((\mathsf{t}', \mathcal{B}) \in \xi) \wedge (((\sigma, \Sigma), (\sigma', \Sigma')) \models \langle \mathcal{B}_{\mathsf{t}'} \rangle * \mathsf{Id})\}$ 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 $k = 0$ then either $ws' <_{\mathcal{H}+1} ws$, or $ws' = ws$ and $\xi_d = \emptyset$.

Proof: From $\mathfrak{L}, \mathcal{D}, R, G \models_{\mathsf{t}} (C_1, \sigma) \preceq (\mathbb{C}, \Sigma) \diamond (u, \mathbb{ws}_1, ws_1, w_1, \mathcal{H}) \Downarrow_{\xi_1} p$, we know there exist u', $\mathbb{ws}_1'$, ws_1', w_1', ξ_d' and ξ_1' such that
- (A) $\mathfrak{L}, \mathcal{D}, R, G \models_{\mathsf{t}} (C_1, \sigma') \preceq (\mathbb{C}, \Sigma') \diamond (u, \mathbb{ws}_1', ws_1', w_1', \mathcal{H}) \Downarrow_{\xi_1'} p$, and
- (B) $\xi_d' = \{(\mathsf{t}', \mathcal{B}) \mid ((\mathsf{t}', \mathcal{B}) \in \xi_1) \wedge (((\sigma, \Sigma), (\sigma', \Sigma')) \models \langle \mathcal{B}_{\mathsf{t}'} \rangle * \mathsf{Id})\}$ 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 $k = 0$ then either $ws_1' <_{\mathcal{H}} ws_1$, or $ws_1' = ws_1$ and $\xi_d = \emptyset$.

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
$$\xi_0' = \{(\mathsf{t}'', \mathcal{B}) \mid ((\mathsf{t}'', \mathcal{B}) \neq \mathsf{t}) \wedge ((\sigma', \Sigma') \models \mathsf{Enabled}(\mathcal{B}_{\mathsf{t}''}) * \mathsf{true})\},$$
$$\xi_d = \{(\mathsf{t}', \mathcal{B}) \mid ((\mathsf{t}', \mathcal{B}) \in \xi) \wedge (((\sigma, \Sigma), (\sigma', \Sigma')) \models \langle \mathcal{B}_{\mathsf{t}'} \rangle * \mathsf{Id})\} \text{ and}$$
$$\xi_d'' = \{(\mathsf{t}', \mathcal{B}) \mid ((\mathsf{t}', \mathcal{B}) \in \xi_0) \wedge (((\sigma, \Sigma), (\sigma', \Sigma')) \models \langle \mathcal{B}_{\mathsf{t}'} \rangle * \mathsf{Id})\}.$$
Since $\mathsf{Enabled}(\mathcal{D}) \Rightarrow I$, $\mathcal{D}' \leqslant \mathcal{D}$ and $\mathsf{wffAct}(R, \mathcal{D}')$, we can prove:
$$k = 0 \implies \xi_0 \backslash \xi_d'' \subseteq \xi_0' .$$

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 $\mathrm{t}' \in \xi_0$ and $((\sigma, \Sigma), (\sigma', \Sigma')) \models \langle \mathcal{D}_{\mathrm{t}'} \rangle * \mathsf{Id}$. Since $\mathcal{D}' \leqslant \mathcal{D}$, we can prove

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

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

$$k'_s < k_s \,.$$

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

- 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
$$\mathfrak{L}, \mathcal{D}, R, G \models_{\mathrm{t}} (C_1; \mathsf{while}\ (B)\{C\}, \sigma') \preceq (\mathbb{C}, \Sigma') \diamond (u, \mathbb{ws}', ws', w', \mathcal{H}+1) \Downarrow_{\xi'} p \wedge \neg B \,.$$
If $k = 0$, we know $\xi \backslash \xi_d \subseteq \xi'$, $\mathbb{ws}' \preceq_{\mathcal{H}+1} \mathbb{ws}$ 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 $(\sigma', \Sigma', i) \models Q_{\mathrm{t}} * \mathsf{true}$. 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
$$\mathfrak{L}, \mathcal{D}, R, G \models_{\mathrm{t}} (C_1; \mathsf{while}\ (B)\{C\}, \sigma') \preceq_i (\mathbb{C}, \Sigma') \diamond (u, \mathbb{ws}', w', \mathcal{H}+1) \Downarrow_{\xi'} p \wedge \neg B \,.$$
If $k = 0$, we know $\xi \backslash \xi_d \subseteq \xi'$, $w' = w$ and $\mathbb{ws}'_1 \preceq_{\mathcal{H}+1} \mathbb{ws}_1$.
  - 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 $(\sigma, \Sigma, i) \models Q_{\mathrm{t}} * \mathsf{true}$ and $\mathbb{ws} = ((0,0),0) :: \mathsf{inchead}(\mathbb{ws}_1, ((k_s, w_1), 1))$, we know: if $k = 0$, $\mathbb{ws}' \preceq_{\mathcal{H}+1} \mathbb{ws}$. 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.3.2 The LIVE-FRM *rule.*

Lemma B.11 (LIVE-FRM-Sound). *If*

(1) $\mathfrak{L}', \mathcal{D}', R, G \models \{p\} C \{q\}$.
(2) $\mathcal{D} = \mathcal{D}' \wedge \mathcal{D}''$; $\mathfrak{L}' \leqslant \mathfrak{L}$;
(3) $I \triangleright \{R, G\}$; $p \Rightarrow J * \mathsf{true}$; $J \Rightarrow I$; $\mathsf{Sta}(J, R \vee G)$;
(4) $\mathit{layGt}(\mathfrak{L}, \mathcal{D}'', J, \mathcal{D}')$;
(5) for any t *and* t' *such that* $\mathrm{t} \neq \mathrm{t}'$, *we have* $G_{\mathrm{t}} \Rightarrow R_{\mathrm{t}'}$;

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

Proof. Let $\mathcal{H} = \mathsf{height}(C)$, $\mathbb{ws} = ((0,0),1)$ and $ws = ((0,0),1)$ and $\xi = \emptyset$.
For any σ, Σ, u, w and $\mathbb{C}$, if $((\sigma, \Sigma), (u, w), \mathbb{C}) \models p_{\mathrm{t}}$, we have

$$\mathfrak{L}', \mathcal{D}', R, G \models_{\mathrm{t}} (C, \sigma) \preceq (\mathbb{C}, \Sigma) \diamond (u, \mathbb{ws}, ws, w, \mathcal{H}) \Downarrow_{\xi} q \,.$$

To prove

$$\mathfrak{L}, \mathcal{D}, R, G \models_{\mathrm{t}} (C, \sigma) \preceq (\mathbb{C}, \Sigma) \diamond (u, \mathbb{ws}, ws, w, \mathcal{H}) \Downarrow_{\xi} q \,.$$

We prove a generalized lemma.

For any σ, Σ, u, w, $\mathbb{C}$, $\mathbb{ws}$, ws, ξ, if $((\sigma,\Sigma),(u,w),\mathbb{C}) \models J * \mathsf{true}$, and

$$\mathfrak{L}', \mathcal{D}', R, G \models_{\mathsf{t}} (C,\sigma) \preceq (\mathbb{C},\Sigma) \diamond (u, \mathbb{ws}, ws, w, \mathcal{H}) \Downarrow_{\xi} \ q\,.$$

then

$$\mathfrak{L}, \mathcal{D}, R, G \models_{\mathsf{t}} (C,\sigma) \preceq (\mathbb{C},\Sigma) \diamond (u, \mathbb{ws}, ws, w, \mathcal{H}) \Downarrow_{\xi} \ q\,.$$

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

(1) if $\delta = \{\mathcal{B} \mid (\mathcal{B} \leqslant \mathcal{D}) \wedge ((\sigma,\Sigma) \models \mathsf{Enabled}(\mathcal{B}_{\mathsf{t}}) * \mathbf{true})\}$, then for any $(\mathsf{t}', \mathcal{B}') \in \xi$, $\mathsf{t}' \neq \mathsf{t}$ and there exists $\mathfrak{S}'$ such that
   (a) there exists $\mathfrak{S}''$, $(\sigma,\Sigma) = \mathfrak{S}' \uplus \mathfrak{S}''$ and $\mathfrak{S}' \models \mathsf{Enabled}(\mathcal{B}'_{\mathsf{t}'})$
   (b) $\mathfrak{L}(\mathfrak{S}', \mathcal{B}')$ is defined and
   (c) for any $\mathcal{B} \in \delta, \mathfrak{L}(\mathfrak{S}', \mathcal{B}) > \mathfrak{L}(\mathfrak{S}', \mathcal{B}')$

   Proof: (a)(b) are immediately true from premise.
   Let $\delta' = \{\mathcal{B} \mid (\mathcal{B} \leqslant \mathcal{D}') \wedge ((\sigma,\Sigma) \models \mathsf{Enabled}(\mathcal{B}_{\mathsf{t}}) * \mathbf{true})\}$.
   From
   $$\mathfrak{L}', \mathcal{D}', R, G \models_{\mathsf{t}} (C,\sigma) \preceq (\mathbb{C},\Sigma) \diamond (u, \mathbb{ws}, ws, w, \mathcal{H}) \Downarrow_{\xi} \ q\,.$$
   we know for any $\mathcal{B}_1 \in \delta'$, $\mathfrak{L}'(\mathfrak{S}', \mathcal{B}_1) > \mathfrak{L}'(\mathfrak{S}', \mathcal{B}')$.
   From $\mathfrak{L}' \leqslant \mathfrak{L}$, we know $\mathfrak{L}(\mathfrak{S}', \mathcal{B}_1) > \mathfrak{L}(\mathfrak{S}', \mathcal{B}')$
   Let $\delta'' = \{\mathcal{B} \mid (\mathcal{B} \leqslant \mathcal{D}'') \wedge ((\sigma,\Sigma) \models \mathsf{Enabled}(\mathcal{B}_{\mathsf{t}}) * \mathbf{true})\}$, from $\mathsf{layGt}(\mathfrak{L}, \mathcal{D}'', J, \mathcal{D}')$, we know for any $\mathcal{B}_2 \in \delta''$, $\mathfrak{L}(\mathfrak{S}', \mathcal{B}_2) > \mathfrak{L}(\mathfrak{S}', \mathcal{B}')$.
   Thus we are done.

(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) $\mathbb{ws} = ((0,0),0)$ and $ws = (0,0)$ and $\xi = \emptyset$, and
   (d) $((\sigma,\Sigma),(\sigma,\Sigma'),0) \models G_{\mathsf{t}} * \mathsf{True}$.

   Proof: From premise we know they are true.

(3) 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) $((\sigma,\Sigma'),(u,w),\mathbf{skip}) \models q_{\mathsf{t}}$ and $[\![E]\!]_{\sigma.s} = [\![\mathbb{E}]\!]_{\Sigma'.s}$, and
   (c) $((\sigma,\Sigma),(\sigma,\Sigma'),0) \models G_{\mathsf{t}} * \mathsf{True}$.

   Proof:
   From premise we know they are true.

(4) For any σ_F, $(C, \sigma \uplus \sigma_F) \not\longrightarrow_{\mathsf{t}} \mathbf{abort}$.
   Proof:
   From premise we know they are true.

(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) $\mathfrak{L}, \mathcal{D}, R, G \models_{\mathsf{t}} (C',\sigma') \preceq (\mathbb{C}',\Sigma') \diamond (u', \mathbb{ws}', ws', w', \mathcal{H}) \Downarrow_{\xi'} q$, 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) $ws' <_{\mathcal{H}} ws$,
or $ws' = ws$ and $\xi \neq \emptyset$ and $\xi \subseteq \xi'$

Proof:
From $\mathrm{Sta}(J, R \vee G)$ and $((\sigma, \Sigma), (\sigma', \Sigma'), k) \models G_{\mathsf{t}} * \mathsf{True}$, we know $(\sigma', \Sigma') \models J * \mathsf{true}$.
From premise we know (a)(b)(d)(e)(f) are true.
By the co-indution hypothesis we know

$$\mathfrak{L}, \mathcal{D}, R, G \models_{\mathsf{t}} (C', \sigma') \preceq (\mathbb{C}', \Sigma') \diamond (u', \mathbb{ws}', ws', w', \mathcal{H}) \Downarrow_{\xi'} q$$

(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) $\mathfrak{L}, \mathcal{D}, R, G \models_{\mathsf{t}} (C, \sigma') \preceq (\mathbb{C}, \Sigma') \diamond (u', \mathbb{ws}', ws', w', \mathcal{H}) \Downarrow_{\xi'} q$, and
(b) $\xi_d = \{(\mathsf{t}', \mathcal{B}) \mid ((\mathsf{t}', \mathcal{B}) \in \xi) \wedge (((\sigma, \Sigma), (\sigma', \Sigma')) \models \langle \mathcal{B}_{\mathsf{t}'} \rangle * \mathsf{Id})\}$ 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 $k = 0$, then either $ws' <_{\mathcal{H}} ws$, or $ws' = ws$ and $\xi_d = \emptyset$.

Proof:
From $\mathrm{Sta}(J, R \vee G)$ and $((\sigma, \Sigma), (\sigma', \Sigma'), k) \models R_{\mathsf{t}} * \mathsf{Id}$, we know $(\sigma', \Sigma') \models J * \mathsf{true}$.
From premise, we know (a)(b)(d)(e)(f) are true.
By the co-indution hypothesis we know

$$\mathfrak{L}, \mathcal{D}, R, G \models_{\mathsf{t}} (C, \sigma') \preceq (\mathbb{C}, \Sigma') \diamond (u', \mathbb{ws}', ws', w', \mathcal{H}) \Downarrow_{\xi'} q$$

Thus we are done.

□

B.4 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 $\Delta_{\mathtt{t}}(\Sigma)$, another thread $\mathtt{t}'$ is possible to make stuttering steps.

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

Definition B.12 (Common Simulation for Object). $\Pi \precsim_P \Pi'$ iff there exists Δ, $\mathfrak{L}$, $\mathcal{D}$, R and G such that $\mathfrak{L}, \mathcal{D}, R, G \models^{\Delta} \{P\}\Pi \precsim \Pi'$ holds.

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

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

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

(1) Suppose $\sigma = (s, h)$, and $\delta = \{\mathcal{B} \mid (\mathcal{B} \leqslant \mathcal{D}) \wedge ((\sigma, \Sigma) \models \mathsf{Enabled}(\mathcal{B}_{\mathtt{t}}) * \mathbf{true})\}$, then
   (a) $\mathtt{tids}(\xi) \subseteq s(\mathtt{TIDS})$ and
   (b) for any $(\mathtt{t}', \mathcal{B}') \in \xi$, $\mathtt{t}' \neq \mathtt{t}$ and there exists $\mathfrak{S}'$ such that
      (b1) there exists $\mathfrak{S}''$, $(\sigma, \Sigma) = \mathfrak{S}' \uplus \mathfrak{S}''$ and $\mathfrak{S}' \models \mathsf{Enabled}(\mathcal{B}'_{\mathtt{t}'})$
      (b2) $\mathfrak{L}(\mathfrak{S}', \mathcal{B}')$ is defined and
      (b3) for any $\mathcal{B} \in \delta$, $\mathfrak{L}(\mathfrak{S}', \mathcal{B}) > \mathfrak{L}(\mathfrak{S}', \mathcal{B}')$
(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) $(\mathbb{C}, \Sigma \uplus \Sigma_F) \longrightarrow^{*}_{\mathtt{t}} (\mathbf{return}\ \mathbb{E}, \Sigma' \uplus \Sigma_F)$, and
   (b) $(\sigma, \Sigma') \models Q_{\mathtt{t}}$ and $[\![E]\!]_{\sigma.s} = [\![\mathbb{E}]\!]_{\Sigma'.s}$, and
   (c) $((\sigma, \Sigma), (\sigma, \Sigma'), 0) \models G_{\mathtt{t}} * \mathsf{True}$.
(3) For any σ_F, $(C, \sigma \uplus \sigma_F) \not\longrightarrow_{\mathtt{t}}$ **abort**.
(4) For any C', σ'', σ_F and Σ_F, if $(C, \sigma \uplus \sigma_F) \longrightarrow_{\mathtt{t}} (C', \sigma'')$ and $\Sigma \perp \Sigma_F$, then there exist σ', n, $\mathbb{C}'$, Σ', k, $\mathbb{M}'$, M' and ξ' such that
   (a) $\sigma'' = \sigma' \uplus \sigma_F$, and
   (b) $(\mathbb{C}, \Sigma \uplus \Sigma_F) \longrightarrow^{n}_{\mathtt{t}} (\mathbb{C}', \Sigma' \uplus \Sigma_F)$; and
   if $k > 0$, then there exist n_1, n_2 and $\mathbb{C}''$ such that $n = n_1 + n_2 > 0$ and $(\mathbb{C}, \Sigma \uplus \Sigma_F) \longrightarrow^{n_1}_{\mathtt{t}} (\mathbb{C}'', \Delta_{\mathtt{t}}(\Sigma) \uplus \Sigma_F)$ and
   $(\mathbb{C}'', \Delta_{\mathtt{t}}(\Sigma) \uplus \Sigma_F) \longrightarrow^{n_2}_{\mathtt{t}} (\mathbb{C}', \Sigma' \uplus \Sigma_F)$; and
   (c) $\mathfrak{L}, \mathcal{D}, R, G \models^{\Delta}_{\mathtt{t}} (C', \sigma') \precsim (\mathbb{C}', \Sigma') \diamond (\mathbb{M}', M') \Downarrow_{\xi'} Q$, and
   (d) $((\sigma, \Sigma), (\sigma', \Sigma'), k) \models G_{\mathtt{t}} * \mathsf{True}$, 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) either $M' < M$,
   or $M' = M$ and $\xi \neq \emptyset$ and $\xi \subseteq \xi'$.
(5) For any k, σ' and Σ', if $((\sigma, \Sigma), (\sigma', \Sigma'), k) \models R_{\mathtt{t}} * \mathsf{Id}$, then there exist $\mathbb{M}'$, M', ξ_d and ξ' such that
   (a) $\mathfrak{L}, \mathcal{D}, R, G \models^{\Delta}_{\mathtt{t}} (C, \sigma') \precsim (\mathbb{C}, \Sigma') \diamond (\mathbb{M}', M') \Downarrow_{\xi'} Q$, and
   (b) $\xi_d = \{(\mathtt{t}', \mathcal{B}) \mid ((\mathtt{t}', \mathcal{B}) \in \xi) \wedge (((\sigma, \Sigma), (\sigma', \Sigma')) \models \langle \mathcal{B}_{\mathtt{t}'} \rangle * \mathsf{Id})\}$ and $(k = 0 \Longrightarrow \xi \backslash \xi_d \subseteq \xi')$, and
   (c) if $k > 0$, then for any $\mathtt{t}' \neq \mathtt{t}$ and $\Sigma_F \perp \Sigma$ we have $(\mathbb{C}, \Delta_{\mathtt{t}'}(\Sigma) \uplus \Sigma_F) \longrightarrow_{\mathtt{t}} (\mathbb{C}, \Delta_{\mathtt{t}'}(\Sigma) \uplus \Sigma_F)$;
   otherwise, $\mathbb{M}' < \mathbb{M}$, or $\mathbb{M}' = \mathbb{M}$ and $\xi_d = \emptyset$; and
   (d) if $k = 0$, then either $M' < M$ or $M' = M$ and $\xi_d = \emptyset$.

B.4.1 Instantiating the Common Simulation.

Lemma B.13 (⑦ in Fig. 25). *Suppose* $R \Rrightarrow \lfloor R \rfloor_0$ *and* $G \Rrightarrow \lfloor G \rfloor_0$. *If* $\mathfrak{L}, \mathcal{D}, R, G \models \{P\}\Pi : \Gamma$, *then* $\mathfrak{L}, \mathcal{D}, R, G \models^{\emptyset} \{P\}\Pi \precsim \Gamma$.

Proof. For any $f \in dom(\Pi)$, 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 $\mathfrak{L}, \mathcal{D}, R, G \models \{P\}\Pi : \Gamma$, we know: there exist three well-founded metrics u, $\mathbb{M}$ and M and a set $\xi \in \mathscr{P}(\mathit{ThrdID} \times \mathit{BaseDAct})$ such that

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

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

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

We only need to prove the following:

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

By co-induction.

(1) Suppose $\sigma = (s, h)$, and $\delta = \{\mathcal{B} \mid (\mathcal{B} \leqslant \mathcal{D}) \wedge ((\sigma, \Sigma) \models \mathsf{Enabled}(\mathcal{B}_{\mathsf{t}}) * \mathbf{true})\}$, then
   (a) $\mathtt{tids}(\xi) \subseteq s(\mathtt{TIDS})$ and
   (b) for any $(\mathsf{t}', \mathcal{B}') \in \xi$, $\mathsf{t}' \neq \mathsf{t}$ and there exists $\mathfrak{S}'$ such that
      (b1) there exists $\mathfrak{S}''$, $(\sigma, \Sigma) = \mathfrak{S}' \uplus \mathfrak{S}''$ and $\mathfrak{S}' \models \mathsf{Enabled}(\mathcal{B}'_{\mathsf{t}'})$
      (b2) $\mathfrak{L}(\mathfrak{S}', \mathcal{B}')$ is defined and
      (b3) for any $\mathcal{B} \in \delta$, $\mathfrak{L}(\mathfrak{S}', \mathcal{B}) > \mathfrak{L}(\mathfrak{S}', \mathcal{B}')$

   Proof: Immediate.

(2) If $C = \mathbf{E}[\,\mathbf{return}\ E\,]$, then for any Σ_F such that $\Sigma \perp \Sigma_F$, 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 $\Sigma \perp \Sigma_F$, then there exist σ', n, $\mathbb{C}'$, Σ', $\mathbb{M}'$, M' and ξ' such that
   (a) $\sigma'' = \sigma' \uplus \sigma_F$, and
   (b) $(\mathbb{C}, \Sigma \uplus \Sigma_F) \longrightarrow^{n}_{\mathsf{t}} (\mathbb{C}', \Sigma' \uplus \Sigma_F)$; and
   (c) $\mathfrak{L}, \mathcal{D}, R, G \models^{\emptyset}_{\mathsf{t}} (C', \sigma') \precsim (\mathbb{C}', \Sigma') \diamond (\mathbb{M}', M') \Downarrow_{\xi'} Q$, 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) either $M' < M$,
      or $M' = M$ and $\xi \neq \emptyset$ and $\xi \subseteq \xi'$

   Proof: Since $\mathfrak{L}, \mathcal{D}, R, G \models_{\mathsf{t}} (C, \sigma) \preceq (\mathbb{C}, \Sigma) \diamond ((u, \mathbb{M}), M) \Downarrow_{\xi} Q$, 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) $\mathfrak{L}, \mathcal{D}, R, G \models_{\mathsf{t}} (C', \sigma') \preceq (\mathbb{C}', \Sigma') \diamond ((u', \mathbb{M}'), M') \Downarrow_{\xi'} Q$, 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) either $M' < M$,
or $M' = M$ and $\xi \neq \emptyset$ and $\xi \subseteq \xi'$

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

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

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) $\mathfrak{L}, \mathcal{D}, R, G \models^{\emptyset}_{\mathsf{t}} (C, \sigma') \precsim (\mathbb{C}, \Sigma') \diamond (\mathbb{M}', M') \Downarrow_{\xi'} Q$, and
(b) $\xi_d = \{(\mathsf{t}', \mathcal{B}) \mid ((\mathsf{t}', \mathcal{B}) \in \xi) \wedge (((\sigma, \Sigma), (\sigma', \Sigma')) \models \langle \mathcal{B}_{\mathsf{t}'} \rangle * \mathsf{Id})\}$ and $(k = 0 \Longrightarrow \xi \backslash \xi_d \subseteq \xi')$, and
(c) if $k > 0$, then for any $\mathsf{t}' \neq \mathsf{t}$ and $\Sigma_F \perp \Sigma$ 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 $k = 0$, then either $M' < M$ or $M' = M$ and $\xi_d = \emptyset$.

Proof: Since $R \Rightarrow \lfloor R \rfloor_0$, we know

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

Then, since $\mathfrak{L}, \mathcal{D}, R, G \models_{\mathsf{t}} (C, \sigma) \preceq (\mathbb{C}, \Sigma) \diamond (u, \mathbb{M}, M) \Downarrow_{\xi} Q$, we know there exist u', $\mathbb{M}'$, M', ξ_d and ξ' such that
(A) $\mathfrak{L}, \mathcal{D}, R, G \models_{\mathsf{t}} (C, \sigma') \preceq (\mathbb{C}, \Sigma') \diamond (u', \mathbb{M}', M') \Downarrow_{\xi'} Q$, 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

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

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 B.14 (⑧ in Fig. 25). *Suppose* $1 \notin fv(\mathfrak{L}, \mathcal{D}, R, G, P, \Pi, \Gamma)$.
If $\mathfrak{L}, \mathcal{D}, R, G \models \{P\}\Pi : \Gamma$, *then* $\mathfrak{L}, \mathcal{D}, R * [1 = 0], G * [1 = 0] \models^{\Delta_1} \{P * (1 = 0)\}\Pi \precsim \mathsf{wr}_1(\Gamma)$. *Here*

$$\Delta_1(\mathsf{t}, \Sigma) \stackrel{\text{def}}{=} \begin{cases} \Sigma\{1 \rightsquigarrow \mathsf{t}\} & \textit{if } \Sigma(1) = 0 \\ \textit{undefined} & \textit{if } 1 \notin \textit{dom}(\Sigma) \textit{ or } \Sigma(1) \neq 0 \end{cases}$$

Proof. For any $f \in \textit{dom}(\Pi)$, for any σ and Σ, for any t, if $\Pi(f) = (x, C)$, $\Gamma(f) = (y, \langle \mathbb{C} \rangle; \mathbf{return}\ \mathbb{E})$ and $(\sigma, \Sigma) \models P_{\mathsf{t}} * (1 = 0) * \mathsf{own}(x) * \mathsf{own}(y) \wedge (x = y)$, we know there exists Σ_1 such that $\Sigma = \Sigma_1 \uplus \{1 \rightsquigarrow 0\}$ and

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Fig. 26. Useful notations for the abstract code wrapper.

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

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

$$\mathfrak{L}, \mathcal{D}, R * [1 = 0], G * [1 = 0] \models_{\mathsf{t}}^{\Delta_1} (C, \sigma) \precsim (\mathsf{wr}_1(\langle\mathbb{C}\rangle; \textbf{return}\ \mathbb{E}), \Sigma) \diamond (\mathbb{M}, M) \Downarrow_{\xi} (P * (1 = 0) * \mathsf{own}(x) * \mathsf{own}(y)).$$

Fig. 26 gives some useful notations for the abstract code wrapper. We only need to prove the following:

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

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

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

(1) Suppose $\sigma = (s, h)$, and $\delta = \{\mathcal{B} \mid (\mathcal{B} \leqslant \mathcal{D}) \wedge ((\sigma, \Sigma) \models \mathsf{Enabled}(\mathcal{B}_{\mathsf{t}}) * \mathbf{true})\}$, then
  (a) $\mathtt{tids}(\xi) \subseteq s(\mathtt{TIDS})$ and
  (b) for any $(\mathsf{t}', \mathcal{B}') \in \xi$, $\mathsf{t}' \neq \mathsf{t}$ and there exists $\mathfrak{S}'$ such that
    (b1) there exists $\mathfrak{S}''$, $(\sigma, \Sigma) = \mathfrak{S}' \uplus \mathfrak{S}''$ and $\mathfrak{S}' \models \mathsf{Enabled}(\mathcal{B}'_{\mathsf{t}'})$
    (b2) $\mathfrak{L}(\mathfrak{S}', \mathcal{B}')$ is defined and
    (b3) for any $\mathcal{B} \in \delta$, $\mathfrak{L}(\mathfrak{S}', \mathcal{B}) > \mathfrak{L}(\mathfrak{S}', \mathcal{B}')$

  Proof: Immediate.

(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) $(\sigma, \Sigma') \models Q_{\mathsf{t}} * (\mathsf{1} = 0)$ and $[\![E]\!]_{\sigma.s} = [\![\mathbb{E}]\!]_{\Sigma'.s}$, and
  (c) $((\sigma, \Sigma), (\sigma, \Sigma'), 0) \models G_{\mathsf{t}} * [\mathsf{1} = 0] * \mathsf{True}$.

  Proof: Immediate from the premise.

(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 $\Sigma \perp \Sigma_F$, then there exist σ', n, $\mathbb{C}'$, Σ', k, $\mathbb{M}'$, M' and ξ' such that
  (a) $\sigma'' = \sigma' \uplus \sigma_F$, and
  (b) $(\widehat{\mathbb{C}}, \Sigma \uplus \Sigma_F) \longrightarrow_{\mathsf{t}}^{n} (\mathbb{C}', \Sigma' \uplus \Sigma_F)$; and
    if $k > 0$, then there exist n_1, n_2 and $\mathbb{C}''$ such that $n = n_1 + n_2 > 0$ and $(\widehat{\mathbb{C}}, \Sigma \uplus \Sigma_F) \longrightarrow_{\mathsf{t}}^{n_1} (\mathbb{C}'', \Delta_{\mathsf{t}}(\Sigma) \uplus \Sigma_F)$ and
    $(\mathbb{C}'', \Delta_{\mathsf{t}}(\Sigma) \uplus \Sigma_F) \longrightarrow_{\mathsf{t}}^{n_2} (\mathbb{C}', \Sigma' \uplus \Sigma_F)$; and
  (c) $\mathfrak{L}, \mathcal{D}, R * [\mathsf{1} = 0], G * [\mathsf{1} = 0] \models_{\mathsf{t}}^{\Delta_1} (C', \sigma') \precsim (\mathbb{C}', \Sigma') \diamond (\mathbb{M}', M') \Downarrow_{\xi'} (Q * (\mathsf{1} = 0))$, and
  (d) $((\sigma, \Sigma), (\sigma', \Sigma'), k) \models G_{\mathsf{t}} * [\mathsf{1} = 0] * \mathsf{True}$, 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) either $M' < M$,
    or $M' = M$ and $\xi \neq \emptyset$ and $\xi \subseteq \xi'$.

  Proof: For 1, from $\mathfrak{L}, \mathcal{D}, R, G \models_{\mathsf{t}} (C, \sigma) \preceq (\langle\mathbb{C}\rangle; \mathbf{return}\ \mathbb{E}, \Sigma_1) \diamond (u, \mathbb{M}, M) \Downarrow_{\xi} Q$, 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) $(\langle\mathbb{C}\rangle; \mathbf{return}\ \mathbb{E}, \Sigma_1 \uplus \Sigma_F) \longrightarrow_{\mathsf{t}}^{*} (\mathbb{C}_1', \Sigma_1' \uplus \Sigma_F)$, and
  (C1) $\mathfrak{L}, \mathcal{D}, R, G \models_{\mathsf{t}} (C', \sigma') \preceq (\mathbb{C}_1', \Sigma_1') \diamond (u', \mathbb{M}', M') \Downarrow_{\xi'} Q$, and
  (D1) $((\sigma, \Sigma_1), (\sigma', \Sigma_1'), k) \models G_{\mathsf{t}} * \mathsf{True}$, 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) either $M' < M$,
    or $M' = M$ and $\xi \neq \emptyset$ and $\xi \subseteq \xi'$.

  From (B1), by the operational semantics, we know

$$\text{either } \mathbb{C}_1' = (\langle\mathbb{C}\rangle; \mathbf{return}\ \mathbb{E}),\ \text{ or } \mathbb{C}_1' = (\mathbf{return}\ \mathbb{E})\ .$$

(i) $\mathbb{C}_1' = (\langle\mathbb{C}\rangle; \mathbf{return}\ \mathbb{E})$: Let

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

From (E1), we know: $u' \le u$, thus $u' \le u_2$. Then, from (C1), by the co-induction hypothesis, we know

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

From (D1), we know

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

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

(ii) $\mathbb{C}_1' = (\mathbf{return}\ \mathbb{E})$: Let

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

From (C1), by the second goal, we know

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

From (D1), we know

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

From (E1), we know $u' \le u$, thus $u_2' \le u_2$. From (B1), we know

$$(\mathsf{wr}_1'(\langle\mathbb{C}\rangle); \mathsf{wr}_1(\mathbf{return}\ \mathbb{E}), \Sigma \uplus \Sigma_F) \longrightarrow_{\mathsf{t}}^{+} (\mathsf{wr}_1'(\mathbf{return}\ \mathbb{E}), \Sigma' \uplus \Sigma_F)\ .$$

If $k > 0$, let

$$\mathbb{C}'' = (\mathsf{wr}_1''(\langle\mathbb{C}\rangle); \mathsf{wr}_1(\mathbf{return}\ \mathbb{E}))\ .$$

Also we know

$$\Delta_1(\mathsf{t}, \Sigma) = \Sigma_1 \uplus \{\mathsf{l} \rightsquigarrow \mathsf{t}, \mathsf{u1} \rightsquigarrow u_1, \mathsf{u2} \rightsquigarrow u_2\}\ .$$

From (B1), we know

$$(\mathsf{wr}_1'(\langle\mathbb{C}\rangle); \mathsf{wr}_1(\mathbf{return}\ \mathbb{E}), \Sigma \uplus \Sigma_F) \longrightarrow_{\mathsf{t}}^{+} (\mathbb{C}'', \Delta_1(\mathsf{t}, \Sigma) \uplus \Sigma_F) \text{ and}$$
$$(\mathbb{C}'', \Delta_1(\mathsf{t}, \Sigma) \uplus \Sigma_F) \longrightarrow_{\mathsf{t}}^{+} (\mathsf{wr}_1'(\mathbf{return}\ \mathbb{E}), \Sigma' \uplus \Sigma_F)\ .$$

For 2, from $\mathfrak{L}, \mathcal{D}, R, G \models_{\mathsf{t}} (C, \sigma) \le (\mathbf{return}\ \mathbb{E}, \Sigma_1) \diamond (u, \mathbb{M}, M) \Downarrow_{\xi} Q$, 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) $(\mathbf{return}\ \mathbb{E}, \Sigma_1 \uplus \Sigma_F) \longrightarrow_{\mathsf{t}}^{*} (\mathbb{C}_1', \Sigma_1' \uplus \Sigma_F)$, and

(C2) $\mathfrak{L}, \mathcal{D}, R, G \models_{\mathsf{t}} (C', \sigma') \le (\mathbb{C}_1', \Sigma_1') \diamond (u', \mathbb{M}', M') \Downarrow_{\xi'} Q$, and

(D2) $((\sigma, \Sigma_1), (\sigma', \Sigma_1'), k) \models G_{\mathsf{t}} * \mathsf{True}$, 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) either $M' < M$,
or $M' = M$ and $\xi \neq \emptyset$ and $\xi \subseteq \xi'$.

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

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

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

$$\mathfrak{L}, \mathcal{D}, R * [1 = 0], G * [1 = 0] \models_{t}^{\Delta_1} (C', \sigma') \precsim (\mathbb{C}', \Sigma') \diamond (\mathbb{M}', M') \Downarrow_{\xi'} (Q * (1 = 0)) .$$

From (D2), we know

$$((\sigma, \Sigma), (\sigma', \Sigma'), k) \models G_t * [1 = 0] * \mathsf{True} .$$

- 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
$$\Delta_1(t, \Sigma) = \Sigma_1 \uplus \{1 \rightsquigarrow t, u1 \rightsquigarrow u_1, u2 \rightsquigarrow u_2\} .$$
Thus we know
$$(\mathsf{wr}'_1(\textbf{return}\ \mathbb{E}), \Sigma \uplus \Sigma_F) \longrightarrow_t^+ (\mathbb{C}'', \Delta_1(t, \Sigma) \uplus \Sigma_F) \text{ and}$$
$$(\mathbb{C}'', \Delta_1(t, \Sigma) \uplus \Sigma_F) \longrightarrow_t^+ (\mathsf{wr}'_1(\textbf{return}\ \mathbb{E}), \Sigma' \uplus \Sigma_F) .$$
- 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
$$(\mathsf{wr}'_1(\textbf{return}\ \mathbb{E}), \Sigma \uplus \Sigma_F) \longrightarrow_t^* (\mathsf{wr}'_1(\textbf{return}\ \mathbb{E}), \Sigma' \uplus \Sigma_F) .$$

(5) For any k, σ' and Σ', if $((\sigma, \Sigma), (\sigma', \Sigma'), k) \models R_t * [1 = 0] * \mathsf{Id}$, then there exist $\mathbb{M}'$, M', ξ_d and ξ' such that
   (a) $\mathfrak{L}, \mathcal{D}, R * [1 = 0], G * [1 = 0] \models_{t}^{\Delta_1} (C, \sigma') \precsim (\widehat{\mathbb{C}}, \Sigma') \diamond (\mathbb{M}', M') \Downarrow_{\xi'} (Q * (1 = 0))$, and
   (b) $\xi_d = \{(t', \mathcal{B}) \mid ((t', \mathcal{B}) \in \xi) \wedge (((\sigma, \Sigma), (\sigma', \Sigma')) \models \langle \mathcal{B}_{t'} \rangle * \mathsf{Id})\}$ and $(k = 0 \Longrightarrow \xi \backslash \xi_d \subseteq \xi')$, and
   (c) if $k > 0$, then for any $t' \neq t$ and $\Sigma_F \perp \Sigma$ we have $(\widehat{\mathbb{C}}, \Delta_{t'}(\Sigma) \uplus \Sigma_F) \longrightarrow_t (\widehat{\mathbb{C}}, \Delta_{t'}(\Sigma) \uplus \Sigma_F)$; otherwise, $\mathbb{M}' < \mathbb{M}$, or $\mathbb{M}' = \mathbb{M}$ and $\xi_d = \emptyset$; and
   (d) if $k = 0$, then either $M' < M$ or $M' = M$ and $\xi_d = \emptyset$.

   Proof: Since $\{1, u1, u2\} \cap fv(R) = \emptyset$, we know there exists Σ'_1 such that
$$((\sigma, \Sigma_1), (\sigma', \Sigma'_1), k) \models R_t * \mathsf{Id} \ \text{ and } \ \Sigma' = \Sigma'_1 \uplus \{1 \rightsquigarrow 0, u1 \rightsquigarrow u_1, u2 \rightsquigarrow u_2\} .$$
Also we know
$$\Delta_1(t', \Sigma) = \Sigma_1 \uplus \{1 \rightsquigarrow t', u1 \rightsquigarrow u_1, u2 \rightsquigarrow u_2\} .$$
Thus we know
$$(\mathsf{wr}'_1(\langle \mathbb{C} \rangle); \mathsf{wr}_1(\textbf{return}\ \mathbb{E}), \Delta_1(t', \Sigma) \uplus \Sigma_F) \longrightarrow_t (\mathsf{wr}'_1(\langle \mathbb{C} \rangle); \mathsf{wr}_1(\textbf{return}\ \mathbb{E}), \Delta_1(t', \Sigma) \uplus \Sigma_F)$$
,
and
$$(\mathsf{wr}'_1(\textbf{return}\ \mathbb{E}), \Delta_1(t', \Sigma) \uplus \Sigma_F) \longrightarrow_t (\mathsf{wr}'_1(\textbf{return}\ \mathbb{E}), \Delta_1(t', \Sigma) \uplus \Sigma_F) .$$
By the co-induction hypothesis, we can prove the goals.

Thus we are done. □

B.5 Core Proofs: From Common Simulation to Contextual Refinement

Lemma B.15 (Gray Box in Fig. 25). *If*

(1) $\mathfrak{L}, \mathcal{D}, R, G \models^{\Delta} \{P\}\Pi \precsim \Pi'$,

(2) $\forall \mathrm{t}, \mathrm{t}'.\ \mathrm{t} \neq \mathrm{t}' \implies G_{\mathrm{t}} \Rightarrow R_{\mathrm{t}'}$, $\mathit{wffAct}(R, \mathcal{D})$, $P \Rightarrow \neg\mathsf{Enabled}(\mathcal{D})$, $P \vee \mathsf{Enabled}(\mathcal{D}) \Rightarrow I$, $I \triangleright \{R, G\}$,

then $\Pi \sqsubseteq_P \Pi'$.

Proof. From $\mathfrak{L}, \mathcal{D}, R, G \models^{\Delta} \{P\}\Pi \precsim \Pi'$, by Lemma B.17, we get: for any C, for any t,

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

(See Sec. B.5.1 for their definitions.) By Lemma B.18, we know: for any $n, C_1, \ldots, C_n$,

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

(Here the simulation $\precsim$ for the whole programs is defined in Definition B.20.) By the following Lemma B.21, we know

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

(Here the fair refinement $\sqsubseteq$ is defined in Definition B.19.) Thus we get: $\Pi \sqsubseteq_P \Pi'$. □

B.5.1 Lifting to Simulation for Client Threads.

Definition B.16 (Simulation for Thread). $\mathfrak{L}, \mathcal{D}, R, G \models^{\Delta}_{\mathrm{t}} \{P\}(\Pi, C) \precsim (\Pi', \mathbb{C})\{Q\}$ 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 $\xi \in \mathscr{P}(\mathit{ThrdID} \times \mathit{BaseDAct})$ such that

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

Here $\mathfrak{L}, \mathcal{D}, R, G \models^{\Delta}_{\mathrm{t}} (\Pi, C, (\sigma_c, \sigma, \kappa)) \precsim (\Pi', \mathbb{C}, (\Sigma_c, \Sigma, \Bbbk)) \diamond (\mathbb{M}, M) \Downarrow_{\xi} Q$ is co-inductively defined as follows.

Whenever $\mathfrak{L}, \mathcal{D}, R, G \models^{\Delta}_{\mathrm{t}} (\Pi, C, (\sigma_c, \sigma, \kappa)) \precsim (\Pi', \mathbb{C}, (\Sigma_c, \Sigma, \Bbbk)) \diamond (\mathbb{M}, M) \Downarrow_{\xi} Q$ holds, then the following hold:

(1) $\sigma_c = \Sigma_c$; and $(C \neq \mathbf{skip}) \Rightarrow (\mathbb{C} \neq \mathbf{skip})$.

(2) Suppose $\sigma = (s, h)$, and $\delta = \{\mathcal{B} \mid (\mathcal{B} \leqslant \mathcal{D}) \wedge ((\sigma, \Sigma) \models \mathsf{Enabled}(\mathcal{B}_{\mathrm{t}}) * \mathbf{true})\}$, then
   (a) $\mathtt{tids}(\xi) \subseteq s(\mathtt{TIDS})$ and
   (b) for any $(\mathrm{t}', \mathcal{B}') \in \xi$, $\mathrm{t}' \neq \mathrm{t}$ and there exists $\mathfrak{S}'$ such that
      (b1) there exists $\mathfrak{S}''$, $(\sigma, \Sigma) = \mathfrak{S}' \uplus \mathfrak{S}''$ and $\mathfrak{S}' \models \mathsf{Enabled}(\mathcal{B}'_{\mathrm{t}'})$
      (b2) $\mathfrak{L}(\mathfrak{S}', \mathcal{B}')$ is defined and
      (b3) for any $\mathcal{B} \in \delta$, $\mathfrak{L}(\mathfrak{S}', \mathcal{B}) > \mathfrak{L}(\mathfrak{S}', \mathcal{B}')$

(3) If $C = \mathbf{skip}$, then $\mathbb{C} = \mathbf{skip}$ and $(\sigma, \Sigma) \models Q_{\mathrm{t}}$.

(4) If $(C, (\sigma_c, \sigma \uplus \sigma_F, \kappa)) \xrightarrow{e}_{\mathrm{t},\Pi} \mathbf{abort}$ and $\Sigma \bot \Sigma_F$, then
   $e = (\mathrm{t}, \mathbf{clt}, \mathbf{abort})$ and there exists $\mathbb{T}$ such that $e = \mathrm{get\_obsv}(\mathbb{T})$ and $(\mathbb{C}, (\Sigma_c, \Sigma \uplus \Sigma_F, \Bbbk)) \xrightarrow{\mathbb{T}}{}^{*}_{\mathrm{t},\Pi} \mathbf{abort}$.

(5) If $(C, (\sigma_c, \sigma \uplus \sigma_F, \kappa)) \xrightarrow{e}_{\mathrm{t},\Pi} (C', (\sigma'_c, \sigma'', \kappa'))$ 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) $(\mathbb{C}, (\Sigma_c, \Sigma \uplus \Sigma_F, \Bbbk)) \xrightarrow{\mathbb{T}}{}^{n}_{\mathrm{t},\Pi'} (\mathbb{C}', (\sigma'_c, \Sigma' \uplus \Sigma_F, \Bbbk'))$; 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
   $(\mathbb{C}, (\Sigma_c, \Sigma \uplus \Sigma_F, \Bbbk)) \xrightarrow{\mathbb{T}_1}{}^{n_1}_{\mathrm{t},\Pi'} (\mathbb{C}'', (\Sigma_c, \Delta_{\mathrm{t}}(\Sigma') \uplus \Sigma_F, \Bbbk))$ and $(\mathbb{C}'', (\Sigma_c, \Delta_{\mathrm{t}}(\Sigma') \uplus \Sigma_F, \Bbbk)) \xrightarrow{\mathbb{T}_2}$

$\overset{n_2}{\underset{\mathsf{t},\Pi'}{\longrightarrow}}$ $(\mathbb{C}', (\sigma_c', \Sigma' \uplus \Sigma_F, \Bbbk'))$;
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) $\mathfrak{L}, \mathcal{D}, R, G \models^{\Delta}_{\mathsf{t}} (\Pi, C', (\sigma_c', \sigma', \kappa')) \precsim (\Pi', \mathbb{C}', (\sigma_c', \Sigma', \Bbbk')) \diamond (\mathbb{M}', M') \Downarrow_{\xi'} Q$, 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 $(e = (\mathsf{t}, \textbf{obj}))$, then either $M' < M$,
or $M' = M$ and $\xi \neq \emptyset$ and $\xi \subseteq \xi'$

(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) $\mathfrak{L}, \mathcal{D}, R, G \models^{\Delta}_{\mathsf{t}} (\Pi, C, (\sigma_c', \sigma', \kappa)) \precsim (\Pi', \mathbb{C}, (\sigma_c', \Sigma', \Bbbk)) \diamond (\mathbb{M}', M') \Downarrow_{\xi'} Q$, and
(b) $\xi_d = \{(\mathsf{t}', \mathcal{B}) \mid ((\mathsf{t}', \mathcal{B}) \in \xi) \wedge (((\sigma, \Sigma), (\sigma', \Sigma')) \models \langle \mathcal{B}_{\mathsf{t}'} \rangle * \mathsf{Id})\}$ and $(k = 0 \Longrightarrow \xi \backslash \xi_d \subseteq \xi')$, and
(c) if $k > 0$, then for any $\mathsf{t}' \neq \mathsf{t}$, $\Sigma_F \perp \Sigma$ and Σ_c' there exists $\mathbb{T}$ such that $(\mathbb{C}, (\Sigma_c', \Delta_{\mathsf{t}'}(\Sigma) \uplus \Sigma_F, \Bbbk)) \xrightarrow[\mathsf{t},\Pi']{\mathbb{T}} (\mathbb{C}, (\Sigma_c', \Delta_{\mathsf{t}'}(\Sigma) \uplus \Sigma_F, \Bbbk))$;
otherwise, $\mathbb{M}' < \mathbb{M}$, or $\mathbb{M}' = \mathbb{M}$ and $\xi_d = \emptyset$; and
(d) if $k = 0$, then either $M' < M$
or $M' = M$ and $\xi_d = \emptyset$.

Lemma B.17 (Lifting). *If* $dom(\Pi) = dom(\Pi')$ *and* $\mathfrak{L}, \mathcal{D}, R, G \models^{\Delta} \{P\}\Pi \precsim \Pi'$, *then for any* t *and* C, *we have* $\mathfrak{L}, \mathcal{D}, R, G \models^{\Delta}_{\mathsf{t}} \{P\}(\Pi, C) \precsim (\Pi', \mathbb{C})\{P\}$.

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

B.5.2 *Parallel Compositionality.*

Lemma B.18 (Parallel Compositionality).
If there exists $\mathfrak{L}$, R, G, $\mathcal{D}$, P *and* Δ *such that the following hold for any* $\mathsf{t} \in [1..n]$:

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

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 $\mathsf{t} \in [1..n]$:

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

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

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

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

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

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

then

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

Here for any $\mathrm{t} \in [1..n]$, $\mathcal{K}(\mathrm{t}) = \kappa_\mathrm{t}$ and $\mathbb{K}(\mathrm{t}) = \Bbbk_\mathrm{t}$, and the functions $\mathscr{M}$, ζ and $\mathscr{L}$ are defined as follows.

- $dom(\mathscr{M}) = dom(\zeta) = dom(()\mathscr{L}) = \mathsf{activeThrds}(\mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \parallel \ldots \parallel C_n) = \{\mathrm{t} \mid (C_\mathrm{t} \neq \mathbf{skip})\}$.
- For any $\mathrm{t} \in dom(\mathscr{M})$, we have $\mathscr{M}(\mathrm{t}) = (\mathbb{M}_\mathrm{t}, \{\mathrm{t}' \rightsquigarrow M_{\mathrm{t}'} \mid \mathsf{dep}(\mathrm{t}, \zeta, \mathrm{t}')\})$ and $\zeta(\mathrm{t}) = \mathsf{tidset}(\xi_\mathrm{t})$.
- For any $\mathrm{t} \in dom(\mathscr{L})$ and $\mathfrak{S}$, $\mathfrak{S}'$ that $\exists \mathfrak{S}'', \mathfrak{S} = \mathfrak{S}' \uplus \mathfrak{S}''$ and $\mathfrak{S}' \models I$. Let $\delta_\mathrm{t} = \{\mathcal{B} \mid \mathcal{B} \leqslant \mathcal{D} \wedge \mathfrak{S}' \models \mathsf{Enabled}(\mathcal{B}_\mathrm{t})\}$, $TidLayer \stackrel{\text{def}}{=} Nat \mid \mathtt{MAX} \mid \mathtt{UNDEF}$

$$\mathsf{dep}(\mathrm{t}, \zeta, \mathrm{t}') = \begin{cases} \mathrm{t}' \in \zeta(\mathrm{t}) \\ \exists \mathrm{t}''.\mathrm{t}'' \in \zeta(\mathrm{t}) \wedge \mathsf{dep}(\mathrm{t}'', \zeta, \mathrm{t}') \end{cases}$$

$$\mathscr{L}(\mathfrak{S}, \mathrm{t}) = \begin{cases} max & \text{if } \delta_\mathrm{t} = \emptyset \\ \mathsf{minl}(\{\mathscr{L}(\mathfrak{S}', \mathcal{B}) \mid \mathcal{B} \in \delta_\mathrm{t}\}) & \text{if } \delta_\mathrm{t} \neq \emptyset \end{cases}$$

$$\mathsf{minl}(Ls) = \begin{cases} undefined & \text{if } Ls = \emptyset \\ \text{if } L \in Nat \text{ then } L \text{ else } undefined & \text{if } Ls = L \cup Ls' \wedge \mathsf{minl}(Ls') = undefined \\ \text{if } L \in Nat \text{ then } min\{L, n\} \text{ else } n & \text{if } Ls = L \cup Ls' \wedge \mathsf{minl}(Ls') = n \end{cases}$$

$layer1 > layer2$ iff
$layer1 \in Nat, layer2 \in Nat$ *and* $layer1 > layer2$ *or*
$layer1 = max$ *and* $layer2 \in Nat$

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

$$(\mathbb{M}', \mathcal{M}') < (\mathbb{M}, \mathcal{M}) \text{ iff } (\mathbb{M}' < \mathbb{M}) \vee (\mathbb{M}' = \mathbb{M}) \wedge (\mathcal{M}' < \mathcal{M})$$

$$\mathcal{M}' < \mathcal{M} \text{ iff } \exists \mathrm{t}.\ (\mathcal{M}'(\mathrm{t}) < \mathcal{M}(\mathrm{t})) \wedge (\forall \mathrm{t}' \in dom(\mathcal{M}) \backslash \{\mathrm{t}\}.\ \mathcal{M}'(\mathrm{t}') \leq \mathcal{M}(\mathrm{t}'))$$

$$\mathcal{M}' \leq \mathcal{M} \text{ iff } \forall \mathrm{t} \in dom(\mathcal{M}).\ \mathcal{M}'(\mathrm{t}) \leq \mathcal{M}(\mathrm{t})$$

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

(B.3)

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

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

(b) forall $\mathrm{t}' \in \zeta(\mathrm{t})$. $\mathscr{L}((\sigma \uplus (\biguplus_\mathrm{t} \sigma_\mathrm{t}), \Sigma \uplus (\biguplus_\mathrm{t} \Sigma_\mathrm{t})), \mathrm{t}) > \mathscr{L}((\sigma \uplus (\biguplus_\mathrm{t} \sigma_\mathrm{t}), \Sigma \uplus (\biguplus_\mathrm{t} \Sigma_\mathrm{t})), \mathrm{t}')$
Proof: For any $\mathrm{t}' \in \zeta(\mathrm{t})$, we know there exists $\mathcal{B}'$, $(\mathrm{t}', \mathcal{B}') \in \xi_\mathrm{t}$, let $\mathfrak{S} = (\sigma \uplus \sigma_\mathrm{t}, \Sigma \uplus \Sigma_\mathrm{t})$, $\delta_\mathrm{t} = \{\mathcal{B} \mid \mathcal{B} \leqslant \mathcal{D} \wedge \mathfrak{S} \models \mathsf{Enabled}(\mathcal{B}_\mathrm{t}) * \mathsf{true}$, from the premise, we know there exists $\mathfrak{S}'$ such that
(B1) there exists $\mathfrak{S}''$, $(\sigma, \Sigma) = \mathfrak{S}' \uplus \mathfrak{S}''$ and $\mathfrak{S}' \models \mathsf{Enabled}(\mathcal{B}'_{\mathrm{t}'})$
(B2) $\mathfrak{L}(\mathfrak{S}', \mathcal{B}')$ is defined and
(B3) for any $\mathcal{B} \in \delta, \mathfrak{L}(\mathfrak{S}', \mathcal{B}) > \mathfrak{L}(\mathfrak{S}', \mathcal{B}')$
From $\mathcal{B}' \leqslant \mathcal{D}$, $\mathsf{Enabled}(\mathcal{D}) \Rightarrow I$, we know $\mathfrak{S}' \models I$, since $(\sigma, \Sigma) \models I$, $\mathsf{Precise}(I)$, we know $\mathfrak{S}' = (\sigma, \Sigma)$ Let $\mathfrak{S}'' = (\sigma \uplus (\biguplus_\mathrm{t} \sigma_\mathrm{t}), \Sigma \uplus (\biguplus_\mathrm{t} \Sigma_\mathrm{t}))$, for any $\mathfrak{S}'''$ that exists $\mathfrak{S}_1$, $\mathfrak{S}'' = \mathfrak{S}''' \uplus \mathfrak{S}_1$ and $\mathfrak{S}''' \models I$, we know $\mathfrak{S}''' = (\sigma, \Sigma)$. Let $\delta_{\mathrm{t}'} = \{\mathcal{B} \mid \mathcal{B} \leqslant$

$\mathcal{D} \wedge (\sigma, \Sigma) \models \mathsf{Enabled}(\mathcal{B}_{\mathsf{t}'})\}$, we know $\mathcal{B}' \in \xi_{\mathsf{t}'}$, then $\mathscr{L}(\mathfrak{S}'', \mathsf{t}') \leq \mathfrak{L}(\mathfrak{S}', \mathcal{B}')$. if $\delta_{\mathsf{t}} = \emptyset$, then $\mathscr{L}(\mathfrak{S}'', \mathsf{t}) = \mathit{max}$ and $\mathscr{L}(\mathfrak{S}'', \mathsf{t}) > \mathscr{L}(\mathfrak{S}'', \mathsf{t}')$ For any $\mathcal{B} \in \delta_{\mathsf{t}}$, from (B3), then $\mathscr{L}(\mathfrak{S}'', \mathsf{t}) > \mathscr{L}(\mathfrak{S}'', \mathsf{t}')$.

(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 $(\mathbb{W}, \mathbb{S}) \overset{\mathbb{T}}{\longmapsto}{}^{+} (\mathbf{skip}, \mathbb{S}')$.
Proof: By the operational semantics we know: for any $\mathsf{t} \in [1..n]$, we have $C_{\mathsf{t}} = \mathbf{skip}$. 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 have $\mathbb{C}_{\mathsf{t}} = \mathbf{skip}$. Thus we have $(\mathbb{W}, \mathbb{S}) \longmapsto (\mathbf{skip}, \mathbb{S})$.

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

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

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

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

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

(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) $\mathsf{t} = \mathsf{tid}(e)$, $\mathsf{get\_obsv}(e) = \mathsf{get\_obsv}(\mathbb{T})$, $(e = (\mathsf{t}, \mathbf{term})) \Rightarrow (e = \mathsf{last}(\mathbb{T}))$;
   (c) $(W', (\sigma_c', \sigma'', \mathcal{K}')) \preceq (\mathbb{W}', \mathbb{S}') \diamond (\mathscr{M}', \zeta', \mathscr{L})$;
   (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 $\zeta(\mathsf{t}) \subseteq \zeta'(\mathsf{t})$ and for any $\mathsf{t}' \in \{\mathsf{t}' \mid \mathsf{dep}(\mathsf{t}, \zeta, \mathsf{t}')\}$, $\zeta(\mathsf{t}') \subseteq \zeta'(\mathsf{t}')$;
   (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 $\zeta(\mathsf{t}') \subseteq \zeta'(\mathsf{t}')$ and for any $\mathsf{t}'' \in \{\mathsf{t}'' \mid \mathsf{dep}(\mathsf{t}', \zeta, \mathsf{t}'')\}$, $\zeta(\mathsf{t}'') \subseteq \zeta'(\mathsf{t}'')$ and $\neg\mathsf{dep}(\mathsf{t}', \zeta, \mathsf{t})$
   or $\mathscr{M}'(\mathsf{t}') = \mathscr{M}(\mathsf{t}')$ and $\zeta(\mathsf{t}') \subseteq \zeta'(\mathsf{t}')$ and for any $\mathsf{t}'' \in \{\mathsf{t}'' \mid \mathsf{dep}(\mathsf{t}', \zeta, \mathsf{t}'')\}$, $\zeta(\mathsf{t}'') \subseteq \zeta'(\mathsf{t}'')$ and $\mathsf{dep}(\mathsf{t}', \zeta, \mathsf{t})$ and $\zeta(t) \neq \emptyset$.

Proof: By the operational semantics, we know there exist t, C_{t}' and κ_{t}' such that

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

By $\mathfrak{L}, \mathcal{D}, R, G \models^{\Delta}_{\mathsf{t}} (\Pi, C_{\mathsf{t}}, (\sigma_c, \sigma \uplus \sigma_{\mathsf{t}}, \kappa_{\mathsf{t}})) \precsim (\Pi', \mathbb{C}_{\mathsf{t}}, (\sigma_c, \Sigma \uplus \Sigma_{\mathsf{t}}, \mathbb{k}_{\mathsf{t}})) \diamond (\mathbb{M}_{\mathsf{t}}, M_{\mathsf{t}}) \Downarrow_{\xi_{\mathsf{t}}} P$, we know
(A) for any $(\mathsf{t}', \mathcal{B}') \in \xi_{\mathsf{t}}$, there exists $\mathfrak{S}'$ and $\mathfrak{S}''$ such that $(\sigma \uplus \sigma_{\mathsf{t}}, \Sigma \uplus \Sigma_{\mathsf{t}}) = \mathfrak{S}' \uplus \mathfrak{S}''$ and $\mathfrak{S}' \models \mathsf{Enabled}(\mathcal{B}'_{\mathsf{t}'})$

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

$(\mathbb{C}_\mathsf{t}, (\sigma_c, \Sigma \uplus (\biguplus_\mathsf{t} \Sigma_\mathsf{t}), \Bbbk_\mathsf{t})) \xrightarrow{\mathbb{T}_1}{}^{n_1}_{\mathsf{t},\Pi'} (\mathbb{C}''_\mathsf{t}, (\sigma_c, \Delta_\mathsf{t}(\Sigma) \uplus (\biguplus_\mathsf{t} \Sigma_\mathsf{t}), \Bbbk_\mathsf{t}))$ and
$(\mathbb{C}''_\mathsf{t}, (\sigma_c, \Delta_\mathsf{t}(\Sigma) \uplus (\biguplus_\mathsf{t} \Sigma_\mathsf{t}), \Bbbk_\mathsf{t})) \xrightarrow{\mathbb{T}_2}{}^{n_2}_{\mathsf{t},\Pi'} (\mathbb{C}'_\mathsf{t}, (\sigma'_c, \Sigma''' \uplus (\biguplus_{\mathsf{t}'\neq\mathsf{t}} \Sigma_{\mathsf{t}'}), \Bbbk'_\mathsf{t}))$;
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) $\mathfrak{L}, \mathcal{D}, R, G \models^{\Delta}_{\mathsf{t}} (\Pi, C'_\mathsf{t}, (\sigma'_c, \sigma''', \kappa'_\mathsf{t})) \precsim (\Pi', \mathbb{C}'_\mathsf{t}, (\sigma'_c, \Sigma''', \Bbbk'_\mathsf{t})) \diamond (\mathbb{M}'_\mathsf{t}, M'_\mathsf{t}) \Downarrow_{\xi'_\mathsf{t}} P$, 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 $(e = (\mathsf{t}, \textbf{obj}))$, then either $M' < M$,
or $M' = M$ and $\xi \neq \emptyset$ and $\xi \subseteq \xi'$

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

(I) for any $(\mathsf{t}'', \mathcal{B}'') \in \xi_{\mathsf{t}'}$, there exists $\mathfrak{S}'$ and $\mathfrak{S}''$ such that $(\sigma \uplus \sigma_{\mathsf{t}'}, \Sigma \uplus \Sigma_{\mathsf{t}'}) = \mathfrak{S}' \uplus \mathfrak{S}''$ and $\mathfrak{S}' \models \mathsf{Enabled}(\mathcal{B}''_{\mathsf{t}''})$

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

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

(K) $\xi_d = \{(\mathsf{t}'', \mathcal{B}) \mid ((\mathsf{t}'', \mathcal{B}) \in \xi_{\mathsf{t}'}) \wedge (((\sigma \uplus \sigma_{\mathsf{t}'}, \Sigma \uplus \Sigma_{\mathsf{t}'}), (\sigma' \uplus \sigma_{\mathsf{t}'}, \Sigma' \uplus \Sigma_{\mathsf{t}'})) \models \langle \mathcal{B}_{\mathsf{t}''} \rangle * \mathsf{Id})\}$
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 $(\mathbb{C}_{\mathsf{t}'}, (\sigma_c, \Delta_\mathsf{t}(\Sigma) \uplus (\biguplus_\mathsf{t} \Sigma_\mathsf{t}), \Bbbk_{\mathsf{t}'})) \xrightarrow{\mathbb{T}'}{}_{\mathsf{t}',\Pi'} (\mathbb{C}_{\mathsf{t}'}, (\sigma_c, \Delta_\mathsf{t}(\Sigma) \uplus (\biguplus_\mathsf{t} \Sigma_\mathsf{t}), \Bbbk_{\mathsf{t}'}))$,
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 $k = 0$, then either $M' < M$
or $M' = M$ and $\xi_d = \emptyset$.

From (C), we know

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

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

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

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.

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

Then by the co-induction hypothesis, we know

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

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 $\zeta(\mathsf{t}) \subseteq \zeta'(\mathsf{t})$. for any t' that $\mathsf{dep}(\mathsf{t}, \zeta, \mathsf{t}')$, From (M), we have $M'_{\mathsf{t}'} \leq M_{\mathsf{t}'}$.
  - If $M'_{\mathsf{t}'} < M_{\mathsf{t}'}$, $\mathscr{M}'(\mathsf{t}) < \mathscr{M}(\mathsf{t})$
  - If $M'_{\mathsf{t}'} < M_{\mathsf{t}'}$, and $\zeta(\mathsf{t}') \neq \emptyset$ and $\zeta(\mathsf{t}') \subseteq \zeta'(\mathsf{t}')$, we get $\mathscr{M}'(\mathsf{t}) = \mathscr{M}(\mathsf{t})$ and we are done

For any $\mathsf{t}' \in \mathit{dom}(\mathscr{M}) \backslash \{\mathsf{t}\}$, 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 from (K), we know $\xi_{\mathsf{t}'} \subseteq \xi'_{\mathsf{t}'}$. for any t'' that $\mathsf{dep}(\mathsf{t}', \zeta, \mathsf{t}'')$, From (H)(M), we know $M'_{\mathsf{t}''} \leq M_{\mathsf{t}''}$,
  - If $M'_{\mathsf{t}''} < M_{\mathsf{t}''}$, $\mathscr{M}'(\mathsf{t}') < \mathscr{M}(\mathsf{t}')$
  - If $M'_{\mathsf{t}''} < M_{\mathsf{t}''}$, and $\zeta(\mathsf{t}'') \neq \emptyset$ and $\zeta(\mathsf{t}'') \subseteq \zeta'(\mathsf{t}'')$, we get $\mathscr{M}'(\mathsf{t}') = \mathscr{M}(\mathsf{t}')$, if $\mathsf{dep}(\mathsf{t}', \zeta, \mathsf{t})$, we know $\xi_{\mathsf{t}} \neq \emptyset$.

Thus we are done. □

B.5.3 Simulation for Whole Programs and Fair Refinement.

Definition B.19 (Fair refinement). $\{P\}W \sqsubseteq \mathbb{W}$ iff

$$\forall \sigma_c, \sigma, \Sigma.\ (\sigma, \Sigma) \models P \implies \mathcal{O}_{\mathit{fo}}[\![W, (\sigma_c, \sigma, \circledcirc))]\!] \subseteq \mathcal{O}_{\mathit{fo}}[\![\mathbb{W}, (\sigma_c, \Sigma, \circledcirc)]\!]\,.$$

Definition B.20 (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 $\mathscr{M} \in \mathit{ThrdID} \rightharpoonup \mathit{Metric}$,$\zeta \in \mathit{ThrdID} \rightharpoonup \mathscr{P}(\mathit{ThrdID})$ and $\mathscr{L} \in (\mathit{RelMem} \times \mathit{ThrdID}) \rightharpoonup \mathit{TidLayer}$ such that

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

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

(1) (a) $\mathit{dom}(\mathscr{M}) = \mathit{dom}(\zeta) = \mathsf{activeThrds}(W) = \mathsf{activeThrds}(\mathbb{W})$, and $\forall \mathsf{t} \in \mathit{dom}(\zeta).\ \zeta(\mathsf{t}) \subseteq \mathit{dom}(\zeta)$.
   (b) for any $\mathsf{t}' \in \zeta(\mathsf{t}).\ \mathscr{L}((\mathcal{S}.\sigma, \mathbb{S}.\Sigma), \mathsf{t}) > \mathscr{L}((\mathcal{S}.\sigma, \mathbb{S}.\Sigma), \mathsf{t}')$

(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 $(\mathbb{W}, \mathbb{S}) \overset{\mathbb{T}}{\longmapsto}{}^{+} (\mathbf{skip}, \mathbb{S}')$.

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

(4) If $(W, \mathcal{S}) \overset{e}{\longmapsto} (W', \mathcal{S}')$, then
   there exist t, $\mathbb{T}$, $\mathbb{W}'$, $\mathbb{S}'$, $\mathscr{M}'$ and ζ' such that all the following hold:
   (a) $(\mathbb{W}, \mathbb{S}) \overset{\mathbb{T}}{\longmapsto}{}^{*} (\mathbb{W}', \mathbb{S}')$;
   (b) $\mathsf{t} = \mathsf{tid}(e)$, $\mathsf{get\_obsv}(e) = \mathsf{get\_obsv}(\mathbb{T})$, $(e = (\mathsf{t}, \mathbf{term})) \Rightarrow (e = \mathsf{last}(\mathbb{T}))$;
   (c) $(W', \mathcal{S}') \preceq (\mathbb{W}', \mathbb{S}') \diamond (\mathscr{M}', \zeta', \mathscr{L})$;
   (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 $\zeta(\mathsf{t}) \subseteq \zeta'(\mathsf{t})$ and for any $\mathsf{t}' \in \{\mathsf{t}' \mid \mathsf{dep}(\mathsf{t}, \zeta, \mathsf{t}')\}$, $\zeta(\mathsf{t}') \subseteq \zeta'(\mathsf{t}')$;
   (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 $\zeta(\mathsf{t}') \subseteq \zeta'(\mathsf{t}')$ and for any $\mathsf{t}'' \in \{\mathsf{t}'' \mid \mathsf{dep}(\mathsf{t}', \zeta, \mathsf{t}'')\}$, $\zeta(\mathsf{t}'') \subseteq \zeta'(\mathsf{t}'')$ and $\neg\mathsf{dep}(\mathsf{t}', \zeta, \mathsf{t})$

$$(\textbf{let}\ \Pi\ \textbf{in}\ C_1 \parallel \ldots C_{\mathrm{t}} \ldots \parallel C_n)|_{\mathrm{t}} \stackrel{\text{def}}{=} C_{\mathrm{t}}$$

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

$$\mathsf{tidset}(\mathcal{E}) \stackrel{\text{def}}{=} \begin{cases} \emptyset & \text{if } \mathcal{E} = \epsilon \\ \{\mathsf{tid}(e)\} \cup \mathsf{tidset}(\mathcal{E}') & \text{if } \mathcal{E} = e :: \mathcal{E}' \end{cases}$$

Fig. 27. Auxiliary definitions.

or $\mathscr{M}'(\mathrm{t}') = \mathscr{M}(\mathrm{t}')$ and $\zeta(\mathrm{t}') \subseteq \zeta'(\mathrm{t}')$ and for any $\mathrm{t}'' \in \{\mathrm{t}'' \mid \mathsf{dep}(\mathrm{t}', \zeta, \mathrm{t}'')\}$, $\zeta(\mathrm{t}'') \subseteq \zeta'(\mathrm{t}'')$ and $\mathsf{dep}(\mathrm{t}', \zeta, \mathrm{t})$ and $\zeta(t) \neq \emptyset$.

Lemma B.21 (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 B.21, we introduce the notion of scheduler χ, which is similar to the event trace $\mathcal{E}$ 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 $\mathcal{E}$ is not sufficient to determine a unique execution if we allow non-deterministic instructions such as `x := rand()`.

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

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 $(W, \mathcal{S}) \stackrel{e}{\longmapsto} (W', \mathcal{S}')$ in Fig. **??** to $(W, \mathcal{S}) \stackrel{\iota}{\longmapsto} (W', \mathcal{S}')$. The definition of the strengthened transition system is similar to the original one and omitted here. Then $(W, \mathcal{S}) \stackrel{\chi}{\longmapsto}{}^{*} (W', \mathcal{S}')$ represents a zero or multi-step execution of $(W, \mathcal{S})$ that leads to $(W', \mathcal{S}')$ under the scheduler χ. $(W, \mathcal{S}) \stackrel{\chi}{\longmapsto}{}^{\omega} \cdot$ represents an infinite execution with the infinite scheduler χ. We use $e = \lfloor \iota \rfloor$ to remove the additional information in ι, and $\mathcal{E} = \lfloor \chi \rfloor$ to remove the additional information in each event in χ. We also overload all the predicates over $\mathcal{E}$ to be defined over χ. For instance, get_obsv(χ) gives us an observable "scheduler" χ_o, which is the subsequence of χ consisting of externally observable events only.

To facilitate the proofs, we give alternative definitions of $O_{f_\mathcal{O}}$. Fig. 28 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 B.22. $(\mathcal{E}_o \in O_{f_\mathcal{O}}[\![W, \mathcal{S}]\!]) \iff (\exists \chi, \chi_o.\ (\chi \models O^{\mathrm{co}}_{f_\mathcal{O}}(\lfloor W \rfloor, \mathcal{S}, \chi_o)) \wedge (\lfloor \chi_o \rfloor = \mathcal{E}_o))$.

Lemma B.23. $(\exists \chi.\ \chi \models O^{\mathrm{co}}_{f_\mathcal{O}}(W, \mathcal{S}, \chi_o)) \iff O^{\mathrm{co}}_{f_\mathcal{O}}(W, \mathcal{S}, \chi_o)$.

Lemma B.24. *If* $\mathcal{M} \models O^{\mathrm{m}}_{f_\mathcal{O}}(W, \mathcal{S}, \chi_o)$, *then* $O^{\mathrm{co}}_{f_\mathcal{O}}(W, \mathcal{S}, \chi_o)$.

We also define a simulation between whole programs under an explicit scheduler χ at the low level, $\chi \models (W, \mathcal{S}) \preceq (\mathbb{W}, \mathbb{S}) \diamond \mathcal{M}$, as a bridge that relates the simulation $(W, \mathcal{S}) \preceq (\mathbb{W}, \mathbb{S}) \diamond (\mathscr{M}, \zeta, \mathscr{L})$ in Definition B.20 to the fair refinement.

$$\frac{(W,\mathcal{S}) \overset{\chi}{\longmapsto}{}^{+} \textbf{abort} \qquad \mathsf{get\_obsv}(\chi) = \chi_o}{\chi \models O^{\mathsf{co}}_{f\!\omega}(W,\mathcal{S},\chi_o)} \qquad \frac{(W,\mathcal{S}) \overset{\chi}{\longmapsto}{}^{+} (\textbf{skip}, \_) \qquad \mathsf{get\_obsv}(\chi) = \chi_o}{\chi \models O^{\mathsf{co}}_{f\!\omega}(W,\mathcal{S},\chi_o)}$$

$$\frac{(W,\mathcal{S}) \overset{\chi}{\longmapsto}{}^{+} (W',\mathcal{S}') \qquad \mathsf{get\_obsv}(\chi) = \epsilon \qquad \chi' \models O^{\mathsf{co}}_{f\!\omega}(W',\mathcal{S},\epsilon) \qquad \mathsf{activeThrds}(W) \subseteq \mathsf{tidset}(\chi)}{\chi :: \chi' \models O^{\mathsf{co}}_{f\!\omega}(W,\mathcal{S},\epsilon)}$$

$$\frac{(W,\mathcal{S}) \overset{\chi}{\longmapsto}{}^{+} (W',\mathcal{S}') \qquad \mathsf{get\_obsv}(\chi) = \iota :: \chi_o \qquad \chi' \models O^{\mathsf{co}}_{f\!\omega}(W',\mathcal{S},\chi'_o) \qquad \mathsf{activeThrds}(W) \subseteq \mathsf{tidset}(\chi)}{\chi :: \chi' \models O^{\mathsf{co}}_{f\!\omega}(W,\mathcal{S},e :: \chi_o :: \chi'_o)}$$

(a) with an explicit scheduler

$$\frac{(W,\mathcal{S}) \overset{\chi}{\longmapsto}{}^{+} \textbf{abort} \qquad \mathsf{get\_obsv}(\chi) = \chi_o}{O^{\mathsf{co}}_{f\!\omega}(W,\mathcal{S},\chi_o)} \qquad \frac{(W,\mathcal{S}) \overset{\chi}{\longmapsto}{}^{+} (\textbf{skip}, \_) \qquad \mathsf{get\_obsv}(\chi) = \chi_o}{O^{\mathsf{co}}_{f\!\omega}(W,\mathcal{S},\chi_o)}$$

$$\frac{(W,\mathcal{S}) \overset{\chi}{\longmapsto}{}^{+} (W',\mathcal{S}') \qquad \mathsf{get\_obsv}(\chi) = \epsilon \qquad O^{\mathsf{co}}_{f\!\omega}(W',\mathcal{S},\epsilon) \qquad \mathsf{activeThrds}(W) \subseteq \mathsf{tidset}(\chi)}{O^{\mathsf{co}}_{f\!\omega}(W,\mathcal{S},\epsilon)}$$

$$\frac{(W,\mathcal{S}) \overset{\chi}{\longmapsto}{}^{+} (W',\mathcal{S}') \qquad \mathsf{get\_obsv}(\chi) = \iota :: \chi_o \qquad O^{\mathsf{co}}_{f\!\omega}(W',\mathcal{S},\chi'_o) \qquad \mathsf{activeThrds}(W) \subseteq \mathsf{tidset}(\chi)}{O^{\mathsf{co}}_{f\!\omega}(W,\mathcal{S},e :: \chi_o :: \chi'_o)}$$

(b) the scheduler is implicit

$$\frac{(W,\mathcal{S}) \overset{\chi}{\longmapsto}{}^{+} \textbf{abort} \qquad \mathsf{get\_obsv}(\chi) = \chi_o}{\mathcal{M} \models O^{\mathsf{m}}_{f\!\omega}(W,\mathcal{S},\chi_o)} \qquad \frac{(W,\mathcal{S}) \overset{\chi}{\longmapsto}{}^{+} (\textbf{skip}, \_) \qquad \mathsf{get\_obsv}(\chi) = \chi_o}{\mathcal{M} \models O^{\mathsf{m}}_{f\!\omega}(W,\mathcal{S},\chi_o)}$$

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

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

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

Fig. 28. Co-inductive definitions for generating observable event traces of complete fair executions.

Definition B.25 (Simulation for the whole program with fixed scheduling at the low level). $\chi \models (W, \mathcal{S}) \preceq (\mathbb{W}, \mathbb{S}) \diamond \mathcal{M}$ is co-inductively defined as follows. Here $\mathcal{M} \in \mathit{ThrdID} \rightharpoonup \mathit{Metric}$. Whenever $\chi \models (W, \mathcal{S}) \preceq (\mathbb{W}, \mathbb{S}) \diamond \mathcal{M}$ holds, then the following hold:

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

Lemma B.26. *For any χ, W, $\mathcal{S}$, χ_o, $\mathbb{W}$, $\mathbb{S}$, $\mathscr{M}$ and ξ, if $\chi \models O^{\mathrm{co}}_{f_o}(W, \mathcal{S}, \chi_o)$ and $(W, \mathcal{S}) \preceq (\mathbb{W}, \mathbb{S}) \diamond (\mathscr{M}, \zeta, \mathscr{L})$, then there exists $\mathcal{M}$ such that $\chi \models (W, \mathcal{S}) \preceq (\mathbb{W}, \mathbb{S}) \diamond \mathcal{M}$.*

Lemma B.27. *If $\chi \models O^{\mathrm{co}}_{f_o}(W, \mathcal{S}, \chi_o)$ and $\chi \models (W, \mathcal{S}) \preceq (\mathbb{W}, \mathbb{S}) \diamond \mathcal{M}$, then $\mathcal{M} \models O^{\mathrm{m}}_{f_o}(\mathbb{W}, \mathbb{S}, \chi_o)$.*

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 B.21.

Proof of Lemma B.21. By Lemmas B.22 and B.23, we only need to show: for any σ_c, σ, Σ, i, χ and χ_o, if $(\sigma, \Sigma, i) \models P$ and $\chi \models O^{\mathrm{co}}_{\mathit{fo}}(\lfloor W \rfloor, (\sigma_c, \sigma, \circledcirc), \chi_o)$, then $O^{\mathrm{co}}_{\mathit{fo}}(\lfloor \mathbb{W} \rfloor, (\sigma_c, \Sigma, \circledcirc), \chi_o)$.

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

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

By Lemma B.26, we know there exists $\mathcal{M}$ such that

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

Then by Lemma B.27, we know

$$\mathcal{M} \models O^{\mathrm{m}}_{\mathit{fo}}(\lfloor \mathbb{W} \rfloor, (\sigma_c, \Sigma, \circledcirc), \chi_o).$$

By Lemma B.24, we get

$$O^{\mathrm{co}}_{\mathit{fo}}(\lfloor \mathbb{W} \rfloor, (\sigma_c, \Sigma, \circledcirc), \chi_o).$$

Thus we are done. □

Proof of Lemma B.22. $\mathcal{E}_o \in O_{f\!\!b}[\![W, \mathcal{S}]\!]$ can be unfolded as follows.

$$\begin{aligned}\mathcal{E}_o \in O_{f\!\!b}[\![W, \mathcal{S}]\!] \quad \text{iff} \quad & \exists \mathcal{E}.\ ((\lfloor W \rfloor, \mathcal{S}) \xmapsto{\mathcal{E}}{}^{+} \textbf{abort}) \wedge (\mathcal{E}_o = \mathsf{get\_obsv}(\mathcal{E})) \\ & \vee ((\lfloor W \rfloor, \mathcal{S}) \xmapsto{\mathcal{E}}{}^{+} (\textbf{skip}, \_)) \wedge (\mathcal{E}_o = \mathsf{get\_obsv}(\mathcal{E})) \\ & \vee ((\lfloor W \rfloor, \mathcal{S}) \xmapsto{\mathcal{E}}{}^{\omega} \cdot) \wedge (\mathcal{E}_o = \mathsf{get\_obsv}(\mathcal{E})) \\ & \qquad \wedge (\forall \mathrm{t} \in \mathsf{activeThrds}(\lfloor W \rfloor).\ |(\mathcal{E}|_{\mathrm{t}})| = \omega \vee \mathsf{last}(\mathcal{E}|_{\mathrm{t}}) = (\mathrm{t}, \textbf{term}))\end{aligned}$$

We define $(\chi_o, \chi) \in O_{f\!\!b}[\![W, \mathcal{S}]\!]$ as follows.

$$\begin{aligned}(\chi_o, \chi) \in O_{f\!\!b}[\![W, \mathcal{S}]\!] \quad \text{iff} \quad & ((W, \mathcal{S}) \xmapsto{\chi}{}^{+} \textbf{abort}) \wedge (\chi_o = \mathsf{get\_obsv}(\chi)) \\ & \vee ((W, \mathcal{S}) \xmapsto{\chi}{}^{+} (\textbf{skip}, \_)) \wedge (\chi_o = \mathsf{get\_obsv}(\chi)) \\ & \vee ((W, \mathcal{S}) \xmapsto{\chi}{}^{\omega} \cdot) \wedge (\chi_o = \mathsf{get\_obsv}(\chi)) \\ & \qquad \wedge (\forall \mathrm{t} \in \mathsf{activeThrds}(W).\ |(\chi|_{\mathrm{t}})| = \omega \vee \mathsf{last}(\chi|_{\mathrm{t}}) = (\mathrm{t}, \textbf{term}))\end{aligned}$$

Then $(\mathcal{E}_o \in O_{f\!\!b}[\![W, \mathcal{S}]\!]) \iff (\exists \chi_o, \chi.\ ((\chi_o, \chi) \in O_{f\!\!b}[\![(\lfloor W \rfloor), \mathcal{S}]\!]) \wedge (\lfloor \chi_o \rfloor = \mathcal{E}_o))$. Below we prove

$$(\chi \models O^{\mathrm{co}}_{f\!\!b}(W, \mathcal{S}, \chi_o)) \iff ((\chi_o, \chi) \in O_{f\!\!b}[\![W, \mathcal{S}]\!]) \tag{B.4}$$

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

□

Proof of Lemma B.23. By co-induction. □

Proof of Lemma B.24. We want to prove the following:

$$\forall W, \mathcal{S}, \chi_o.\\ (\,\exists \mathcal{M}.\ (\mathcal{M} \models O^{\mathrm{m}}_{\mathit{fo}}(W, \mathcal{S}, \chi_o))\,) \implies O^{\mathrm{co}}_{\mathit{fo}}(W, \mathcal{S}, \chi_o)$$

By co-induction.

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

Figure 28 defines F and gfp F (i.e., $O^{\mathrm{co}}_{\mathit{fo}}$ at the middle part of the figure). Let

$$S \stackrel{\text{def}}{=} \{(W, \mathcal{S}, \chi_o) \mid \exists \mathcal{M}.\ (\mathcal{M} \models O^{\mathrm{m}}_{\mathit{fo}}(W, \mathcal{S}, \chi_o))\}.$$

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

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

By unfolding S and by inversion over $O^{\mathrm{m}}_{\mathit{fo}}$, we have the following cases.

(1) $(W, \mathcal{S}) \stackrel{\chi}{\longmapsto}{}^{+}$ **abort** and $\mathrm{get\_obsv}(\chi) = \chi_o$:
From the definition of F (at the middle part in Figure 28), we know $(W, \mathcal{S}, \chi_o) \in F(S)$.

(2) $(W, \mathcal{S}) \stackrel{\chi}{\longmapsto}{}^{+}$ (**skip**, _) and $\mathrm{get\_obsv}(\chi) = \chi_o$:
From the definition of F (at the middle part in Figure 28), we know $(W, \mathcal{S}, \chi_o) \in F(S)$.

(3) $\chi_o = \epsilon$, $(W, \mathcal{S}) \stackrel{\chi_x}{\longmapsto}{}^{*} (W_y, \mathcal{S}_y)$, $\mathrm{get\_obsv}(\chi_x) = \epsilon$, $(\mathcal{M}_y \models O^{\mathrm{m}}_{\mathit{fo}}(W_y, \mathcal{S}_y, \epsilon))$, $\mathit{dom}(\mathcal{M}) = \mathrm{activeThrds}(W) \neq \emptyset$ and $\forall \mathrm{t} \in \mathit{dom}(\mathcal{M}) \backslash \mathrm{tidset}(\chi_x).\ \mathcal{M}_y(\mathrm{t}) < \mathcal{M}(\mathrm{t})$:
We want to prove $(\chi, W, \mathcal{S}, \epsilon) \in F(S)$. We have two cases:

(a) $\mathrm{activeThrds}(W) \subseteq \mathrm{tidset}(\chi_x)$:
Since $\mathrm{activeThrds}(W) \neq \emptyset$, we know $(W, \mathcal{S}) \stackrel{\chi_x}{\longmapsto}{}^{+} (W_y, \mathcal{S}_y)$. From $(\mathcal{M}_y \models O^{\mathrm{m}}_{\mathit{fo}}(W_y, \mathcal{S}_y, \epsilon))$, we know $(W_y, \mathcal{S}_y, \epsilon) \in S$. From the definition of F (at the middle part in Figure 28), we know $(W, \mathcal{S}, \epsilon) \in F(S)$.

(b) $\mathit{dom}(\mathcal{M}) \backslash \mathrm{tidset}(\chi_x) \neq \emptyset$:
By transfinite induction over the metric $(\mathit{dom}(\mathcal{M}) \backslash \mathrm{tidset}(\chi_x), \mathcal{M}_y)$.

$$\text{Transfinite Induction Principle:}\\ (\forall M.\ (\forall M'.\ M' < M \implies P(M')) \implies P(M)\,) \implies \forall M.P(M)$$

The order over the metrics, $(\xi_2, \mathcal{M}_2) < (\xi_1, \mathcal{M}_1)$, is defined as follows.
$(\xi_2, \mathcal{M}_2) < (\xi_1, \mathcal{M}_1)$ iff $(\xi_2 \subset \xi_1) \vee (\xi_2 = \xi_1 \neq \emptyset) \wedge (\forall \mathrm{t} \in \xi_2.\ \mathcal{M}_2(\mathrm{t}) < \mathcal{M}_1(\mathrm{t}))$
It is clear that $(\xi_2, \mathcal{M}_2) < (\xi_1, \mathcal{M}_1)$ is a well-founded order.
We view our goal as $\forall \mathcal{M}, \chi_x, \mathcal{M}_y.P(\mathcal{M}, \chi_x, \mathcal{M}_y)$, which is reformulated as follows.

$$\begin{array}{l}\forall \mathcal{M}, \chi_x, \mathcal{M}_y.\\ \quad \forall W_y, \mathcal{S}_y.\\ \quad ((W, \mathcal{S}) \stackrel{\chi_x}{\longmapsto}{}^{*} (W_y, \mathcal{S}_y)) \wedge (\mathrm{get\_obsv}(\chi_x) = \epsilon)\\ \quad \wedge\ (\mathcal{M}_y \models O^{\mathrm{m}}_{\mathit{fo}}(W_y, \mathcal{S}_y, \epsilon)) \wedge (\forall \mathrm{t} \in \mathit{dom}(\mathcal{M}) \backslash \mathrm{tidset}(\chi_x).\ \mathcal{M}_y(\mathrm{t}) < \mathcal{M}(\mathrm{t}))\\ \quad \implies\\ \quad (W, \mathcal{S}, \epsilon) \in F(S)\end{array}$$

By the transfinite induction principle, we only need to prove

$$\begin{array}{l}\forall \mathcal{M}, \chi_x, \mathcal{M}_y.\\ (\forall \mathcal{M}', \chi'_x, \mathcal{M}'_y.\ (\mathit{dom}(\mathcal{M}') \backslash \mathrm{tidset}(\chi'_x), \mathcal{M}'_y) < (\mathit{dom}(\mathcal{M}) \backslash \mathrm{tidset}(\chi_x), \mathcal{M}_y)\\ \implies P(\mathcal{M}', \chi'_x, \mathcal{M}'_y))\\ \implies P(\mathcal{M}, \chi_x, \mathcal{M}_y)\end{array}$$

The proof is by inversion over $(\mathcal{M}_y \models O^{\mathrm{m}}_{fo}(W_y, \mathcal{S}_y, \epsilon))$. We only have two possible cases:

(i) $(W_y, \mathcal{S}_y) \xrightarrow{\chi_y}{}^{+} (\mathbf{skip}, \_)$ and $\mathsf{get\_obsv}(\chi_y) = \epsilon$:
We get $(W, \mathcal{S}) \xrightarrow{\chi_x :: \chi_y}{}^{+} (\mathbf{skip}, \_)$ and $\mathsf{get\_obsv}(\chi_x :: \chi_y) = \epsilon$. By the definition of F (at the middle part in Figure 28), we know $(W, \mathcal{S}, \epsilon) \in F(S)$.

(ii) $(W_y, \mathcal{S}_y) \xrightarrow{\chi'_x}{}^{*} (W'_y, \mathcal{S}'_y)$, $\mathsf{get\_obsv}(\chi'_x) = \epsilon$, $(\mathcal{M}'_y \models O^{\mathrm{m}}_{fo}(W'_y, \mathcal{S}'_y, \epsilon))$, $dom(\mathcal{M}_y) = \mathsf{activeThrds}(W_y) \neq \emptyset$ and $\forall \mathrm{t} \in dom(\mathcal{M}_y) \backslash \mathsf{tidset}(\chi'_x).\ \mathcal{M}'_y(\mathrm{t}) < \mathcal{M}_y(\mathrm{t})$:
We first show

$$(dom(\mathcal{M}_y) \backslash \mathsf{tidset}(\chi_x :: \chi'_x), \mathcal{M}'_y) < (dom(\mathcal{M}) \backslash \mathsf{tidset}(\chi_x), \mathcal{M}_y)\ . \tag{B.5}$$

By the operational semantics, we know

$$dom(\mathcal{M}_y) = \mathsf{activeThrds}(W_y) \subseteq \mathsf{activeThrds}(W) = dom(\mathcal{M}).$$

Since $\mathsf{tidset}(\chi_x) \subseteq \mathsf{tidset}(\chi_x :: \chi'_x)$, we know $dom(\mathcal{M}_y) \backslash \mathsf{tidset}(\chi_x :: \chi'_x) \subseteq dom(\mathcal{M}) \backslash \mathsf{tidset}(\chi_x)$. If $dom(\mathcal{M}_y) \backslash \mathsf{tidset}(\chi_x :: \chi'_x) = dom(\mathcal{M}) \backslash \mathsf{tidset}(\chi_x)$, for any $\mathrm{t} \in dom(\mathcal{M}_y) \backslash \mathsf{tidset}(\chi_x :: \chi'_x)$, we know $\mathrm{t} \in dom(\mathcal{M}_y) \backslash \mathsf{tidset}(\chi'_x)$. Thus $\mathcal{M}'_y(\mathrm{t}) < \mathcal{M}_y(\mathrm{t})$. Thus we have proved (B.5).
By the transfinite induction hypothesis, we have

$$\begin{array}{l}
\forall W'_y, \mathcal{S}'_y. \\
((W, \mathcal{S}) \xrightarrow{\chi_x :: \chi'_x}{}^{*} (W'_y, \mathcal{S}'_y)) \wedge (\mathsf{get_obsv}(\chi_x :: \chi'_x) = \epsilon) \\
\wedge\ (\mathcal{M}'_y \models O^{\mathrm{m}}_{fo}(W'_y, \mathcal{S}'_y, \epsilon)) \wedge (\forall \mathrm{t} \in dom(\mathcal{M}_y) \backslash \mathsf{tidset}(\chi_x :: \chi'_x).\ \mathcal{M}'_y(\mathrm{t}) < \mathcal{M}_y(\mathrm{t})) \\
\Longrightarrow \\
(W, \mathcal{S}, \epsilon) \in F(S)
\end{array}$$

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

(4) $\chi_o = \iota :: \chi_a :: \chi_b$, $(W, \mathcal{S}) \xrightarrow{\chi_x}{}^{+} (W_y, \mathcal{S}_y)$, $\mathsf{get\_obsv}(\chi_x) = \iota :: \chi_a$, $(\mathcal{M}_y \models O^{\mathrm{m}}_{fo}(W_y, \mathcal{S}_y, \chi_b))$, $dom(\mathcal{M}) = \mathsf{activeThrds}(W)$ and $\forall \mathrm{t} \in dom(\mathcal{M}) \backslash \mathsf{tidset}(\chi_x).\ \mathcal{M}_y(\mathrm{t}) < \mathcal{M}(\mathrm{t})$:
We want to prove $(W, \mathcal{S}, \epsilon) \in F(S)$. We have two cases:

(a) $\mathsf{activeThrds}(W) \subseteq \mathsf{tidset}(\chi_x)$:
From $(\mathcal{M}_y \models O^{\mathrm{m}}_{fo}(W_y, \mathcal{S}_y, \chi_b))$, we know $(W_y, \mathcal{S}_y, \chi_b) \in S$. From the definition of F (at the middle part in Figure 28), we know $(W, \mathcal{S}, \chi_o) \in F(S)$.

(b) $dom(\mathcal{M}) \backslash \mathsf{tidset}(\chi_x) \neq \emptyset$:
By transfinite induction over the metric $(dom(\mathcal{M}) \backslash \mathsf{tidset}(\chi_x), \mathcal{M}_y)$.

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.

$$(\xi_2, \mathcal{M}_2) < (\xi_1, \mathcal{M}_1)\ \text{ iff }\ (\xi_2 \subset \xi_1)\ \vee\ (\xi_2 = \xi_1 \neq \emptyset) \wedge (\forall \mathrm{t} \in \xi_2.\ \mathcal{M}_2(\mathrm{t}) < \mathcal{M}_1(\mathrm{t}))$$

It is clear that $(\xi_2, \mathcal{M}_2) < (\xi_1, \mathcal{M}_1)$ is a well-founded order.
We view our goal as $\forall \mathcal{M}, \chi_x, \mathcal{M}_y.P(\mathcal{M}, \chi_x, \mathcal{M}_y)$, which is reformulated as follows.

$$\begin{array}{l}
\forall \mathcal{M}, \chi_x, \mathcal{M}_y. \\
\quad \forall W_y, \mathcal{S}_y, \iota, \chi_a, \chi_b. \\
\quad (\chi_o = \iota :: \chi_a :: \chi_b) \wedge ((W, \mathcal{S}) \xrightarrow{\chi_x}{}^{+} (W_y, \mathcal{S}_y)) \wedge (\mathsf{get_obsv}(\chi_x) = \iota :: \chi_a) \\
\quad \wedge\ (\mathcal{M}_y \models O^{\mathrm{m}}_{fo}(W_y, \mathcal{S}_y, \chi_b)) \wedge (\forall \mathrm{t} \in dom(\mathcal{M}) \backslash \mathsf{tidset}(\chi_x).\ \mathcal{M}_y(\mathrm{t}) < \mathcal{M}(\mathrm{t})) \\
\quad \Longrightarrow \\
\quad (W, \mathcal{S}, \chi_o) \in F(S)
\end{array}$$

By the transfinite induction principle, we only need to prove

$\forall \mathcal{M}, \chi_x, \mathcal{M}_y.$
$(\forall \mathcal{M}', \chi_x', \mathcal{M}_y'.\ (dom(\mathcal{M}')\backslash \mathrm{tidset}(\chi_x'), \mathcal{M}_y') < (dom(\mathcal{M})\backslash \mathrm{tidset}(\chi_x), \mathcal{M}_y)$
$\Longrightarrow P(\mathcal{M}', \chi_x', \mathcal{M}_y'))$
$\Longrightarrow P(\mathcal{M}, \chi_x, \mathcal{M}_y)$

The proof is by inversion over $(\mathcal{M}_y \models O^{\mathrm{m}}_{f\!o}(W_y, \mathcal{S}_y, \chi_b))$. We have four cases:

(i) $(W_y, \mathcal{S}_y) \xmapsto{\chi_y}{}^{+}$ **abort** and $\mathrm{get\_obsv}(\chi_y) = \chi_b$:
We get $(W, \mathcal{S}) \xmapsto{\chi_x :: \chi_y}{}^{+}$ **abort** and $\mathrm{get\_obsv}(\chi_x :: \chi_y) = \chi_o$. By the definition of F (at the middle part in Figure 28), we know $(W, \mathcal{S}, \chi_o) \in F(S)$.

(ii) $(W_y, \mathcal{S}_y) \xmapsto{\chi_y}{}^{+}$ (**skip**, _) and $\mathrm{get\_obsv}(\chi_y) = \chi_b$:
We get $(W, \mathcal{S}) \xmapsto{\chi_x :: \chi_y}{}^{+}$ (**skip**, _) and $\mathrm{get\_obsv}(\chi_x :: \chi_y) = \chi_o$. By the definition of F (at the middle part in Figure 28), we know $(W, \mathcal{S}, \chi_o) \in F(S)$.

(iii) $\chi_b = \epsilon$, $(W_y, \mathcal{S}_y) \xmapsto{\chi_x'}{}^{*} (W_y', \mathcal{S}_y')$, $\mathrm{get\_obsv}(\chi_x') = \epsilon$, $(\mathcal{M}_y' \models O^{\mathrm{m}}_{f\!o}(W_y', \mathcal{S}_y', \epsilon))$, $dom(\mathcal{M}_y) = \mathrm{activeThrds}(W_y) \neq \emptyset$ and $\forall \mathrm{t} \in dom(\mathcal{M}_y)\backslash \mathrm{tidset}(\chi_x').\ \mathcal{M}_y'(\mathrm{t}) < \mathcal{M}_y(\mathrm{t})$:
With (B.5), by the transfinite induction hypothesis, we have

$\forall W_y', \mathcal{S}_y', \iota, \chi_a, \chi_b.$
$(\chi_o = \iota :: \chi_a :: \chi_b) \wedge ((W, \mathcal{S}) \xmapsto{\chi_x :: \chi_x'}{}^{+} (W_y', \mathcal{S}_y')) \wedge (\mathrm{get\_obsv}(\chi_x :: \chi_x') = \iota :: \chi_a)$
$\wedge (\mathcal{M}_y' \models O^{\mathrm{m}}_{f\!o}(W_y', \mathcal{S}_y', \chi_b)) \wedge (\forall \mathrm{t} \in dom(\mathcal{M}_y)\backslash \mathrm{tidset}(\chi_x :: \chi_x').\ \mathcal{M}_y'(\mathrm{t}) < \mathcal{M}_y(\mathrm{t}))$
$\Longrightarrow$
$(W, \mathcal{S}, \chi_o) \in F(S)$
Thus we can get $(W, \mathcal{S}, \chi_o) \in F(S)$.

(iv) $\chi_b = \iota' :: \chi_a' :: \chi_b'$, $(W_y, \mathcal{S}_y) \xmapsto{\chi_x'}{}^{+} (W_y', \mathcal{S}_y')$, $\mathrm{get\_obsv}(\chi_x') = \iota' :: \chi_a'$, $(\mathcal{M}_y' \models O^{\mathrm{m}}_{f\!o}(W_y', \mathcal{S}_y', \chi_b'))$,
$dom(\mathcal{M}_y) = \mathrm{activeThrds}(W_y)$ and $\forall \mathrm{t} \in dom(\mathcal{M}_y)\backslash \mathrm{tidset}(\chi_x').\ \mathcal{M}_y'(\mathrm{t}) < \mathcal{M}_y(\mathrm{t})$:
With (B.5), by the transfinite induction hypothesis, we have

$\forall W_y', \mathcal{S}_y', \iota, \chi_a, \iota', \chi_a', \chi_b'.$
$(\chi_o = \iota :: \chi_a :: \iota' :: \chi_a' :: \chi_b') \wedge ((W, \mathcal{S}) \xmapsto{\chi_x :: \chi_x'}{}^{+} (W_y', \mathcal{S}_y'))$
$\wedge (\mathrm{get\_obsv}(\chi_x :: \chi_x') = \iota :: \chi_a :: \iota' :: \chi_a')$
$\wedge (\mathcal{M}_y' \models O^{\mathrm{m}}_{f\!o}(W_y', \mathcal{S}_y', \chi_b')) \wedge (\forall \mathrm{t} \in dom(\mathcal{M}_y)\backslash \mathrm{tidset}(\chi_x :: \chi_x').\ \mathcal{M}_y'(\mathrm{t}) < \mathcal{M}_y(\mathrm{t}))$
$\Longrightarrow$
$(W, \mathcal{S}, \chi_o) \in F(S)$
Thus we can get $(W, \mathcal{S}, \chi_o) \in F(S)$.

Then we are done. □

Proof of Lemma B.26. First we define

$$\mathsf{roundsub}(\chi,\xi,\chi_x) \text{ iff } (|\chi|\neq\omega)\wedge(\chi_x=\chi)\ \vee\ \exists\chi'.\,(\chi=\chi_x::\chi')\wedge(|\chi_x|\neq\omega)\wedge(\xi\subseteq\mathsf{tidset}(\chi_x))$$

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

$$\mathsf{minPos}(\chi,\xi)\stackrel{\text{def}}{=}\begin{cases}1 & \text{if } \chi=\epsilon\\ 1 & \text{if } \chi=\iota::\chi',|\chi|\neq\omega \text{ and } \mathsf{tid}(\iota)\in\xi\\ 1+\mathsf{minPos}(\chi',\xi) & \text{if } \chi=\iota::\chi',|\chi|\neq\omega \text{ and } \mathsf{tid}(\iota)\notin\xi\end{cases}$$

$$\mathsf{nonblocker}(\mathsf{t},\mathsf{t}_n,\zeta) \text{ iff } (\zeta(\mathsf{t})=\emptyset\wedge\mathsf{t}_n=\mathsf{t})\vee(\zeta(\mathsf{t})\neq\emptyset\wedge\mathsf{dep}(\mathsf{t},\zeta,\mathsf{t}_n)\wedge\zeta(\mathsf{t}_n)=\emptyset).$$

We prove the following (B.6) by inversion over $\chi\models O^{\mathrm{co}}_{f_O}(W,\mathcal{S},\chi_o)$.

If $\chi\models O^{\mathrm{co}}_{f_O}(W,\mathcal{S},\chi_o)$, then there exists χ_x such that $\mathsf{roundsub}(\chi,\mathsf{activeThrds}(W),\chi_x)$. (B.6)

We can prove the following (B.7)

If $(W,\mathcal{S})\preceq(\mathbb{W},\mathbb{S})\diamond(\mathscr{M},\zeta,\mathscr{L})$, then for any $\mathsf{t}\in dom(\zeta)$, there exists t_n such that $\mathsf{nonblocker}(\mathsf{t},\mathsf{t}_n,\zeta)$. (B.7)

We can prove the following (B.8)

If $\chi=\chi_x::\chi_z$, $\chi=\iota::\chi'$, $\chi_x=\iota::\chi_y$, $\chi'=\chi_y::\chi_z$, $\mathsf{t}\notin\xi_n$, $\mathsf{t}=\mathsf{tid}(\lfloor\iota\rfloor)$, $\mathsf{roundsub}(\chi,\mathsf{activeThrds}(W),\chi_x)$, $\xi_n\subseteq\mathsf{activeThrds}(W)$, then $\mathsf{roundsub}(\chi',\xi_n,\chi_y)$. (B.8)

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

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

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

(1) $dom(\mathcal{M})=\mathsf{activeThrds}(W)$.
(2) For any t and $\mathtt{tidset}_n$ such that $\mathsf{t}\in dom(\mathcal{M})$ and $\mathtt{tidset}_n=\{\mathsf{t}_n\mid\mathsf{nonblocker}(\mathsf{t},\mathsf{t}_n,\zeta)\}$, we have $\mathcal{M}(\mathsf{t})=(\mathscr{M}(\mathsf{t}),(\{\mathsf{t}'\rightsquigarrow\zeta(\mathsf{t}')\mid\mathsf{dep}(\mathsf{t},\zeta,\mathsf{t}')\vee\mathsf{t}=\mathsf{t}'\},dom(\zeta)),\mathsf{minPos}(\chi_x,\mathtt{tidset}_n))$.

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

$$(M',(\zeta',\xi'_D),k')<(M,(\zeta,\xi_D),k) \text{ iff}$$
$$(M'<M)\ \vee\ (M'=M)\wedge((\zeta',\xi'_D)<(\zeta,\xi_D))\ \vee\ (M'=M)\wedge((\zeta',\xi'_D)=(\zeta,\xi_D))\wedge(k'<k)$$

$$(\zeta',\xi'_D)<(\zeta,\xi_D) \text{ iff}$$
$$(\exists\mathsf{t}.\,\zeta'(\mathsf{t})\supset\zeta(\mathsf{t}))\wedge(\forall\mathsf{t}.\,\zeta'(\mathsf{t})\supseteq\zeta(\mathsf{t})\wedge\zeta'(\mathsf{t})\subseteq\xi'_D\wedge\zeta(\mathsf{t})\subseteq\xi_D)\wedge(\xi'_D\subseteq\xi_D)$$

$$(\zeta',\xi'_D)=(\zeta,\xi_D) \text{ iff}$$
$$(\forall\mathsf{t}.\,\zeta'(\mathsf{t})=\zeta(\mathsf{t})\wedge\zeta'(\mathsf{t})\subseteq\xi'_D\wedge\zeta(\mathsf{t})\subseteq\xi_D)\wedge(\xi'_D\subseteq\xi_D)$$

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

Next we prove: for any χ, W, $\mathcal{S}$, χ_o, $\mathbb{W}$, $\mathbb{S}$, $\mathscr{M}$, ξ, $\mathcal{M}$ and χ_x, if

(1) $\chi\models O^{\mathrm{co}}_{f_O}(W,\mathcal{S},\chi_o)$;
(2) $(W,\mathcal{S})\preceq(\mathbb{W},\mathbb{S})\diamond(\mathscr{M},\zeta,\mathscr{L})$;

(3) $\mathsf{roundsub}(\chi, \mathsf{activeThrds}(W), \chi_x)$;
$dom(\mathcal{M}) = \mathsf{activeThrds}(W)$; and
for any t and ξ_n such that $\mathsf{t} \in dom(\mathcal{M})$ and $\xi_n = \{\mathsf{t}_n \mid \mathsf{nonblocker}(\mathsf{t}, \mathsf{t}_n, \zeta)\}$, we have $\mathcal{M}(\mathsf{t}) = (\mathscr{M}(\mathsf{t}), (\{\mathsf{t}' \rightsquigarrow \zeta(\mathsf{t}') \mid \mathsf{dep}(\mathsf{t}, \zeta, \mathsf{t}') \vee \mathsf{t} = \mathsf{t}'\}, dom(\zeta)), \mathsf{minPos}(\chi_x, \xi_n))$.

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

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

(1) $dom(\mathcal{M}) = \mathsf{activeThrds}(W) = \mathsf{activeThrds}(\mathbb{W})$.
Proof: From $(W, \mathcal{S}) \preceq (\mathbb{W}, \mathbb{S}) \diamond (\mathscr{M}, \zeta, \mathscr{L})$, 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 $\chi = \epsilon$, then
there exist $\mathbb{T}$ and $\mathbb{S}'$ such that $\mathsf{get\_obsv}(\mathbb{T}) = \epsilon$ and $(\mathbb{W}, \mathbb{S}) \xmapsto{\mathbb{T}}{}^{+} (\mathbf{skip}, \mathbb{S}')$.
Proof: From $(W, \mathcal{S}) \preceq (\mathbb{W}, \mathbb{S}) \diamond (\mathscr{M}, \zeta, \mathscr{L})$, we know there exist $\mathbb{T}$ and $\mathbb{S}'$ such that $\mathsf{get\_obsv}(\mathbb{T}) = \epsilon$ and $(\mathbb{W}, \mathbb{S}) \xmapsto{\mathbb{T}}{}^{+} (\mathbf{skip}, \mathbb{S}')$.

(3) If $(W, \mathcal{S}) \xmapsto{\iota} \mathbf{abort}$ and $\chi = \iota :: \epsilon$, 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 $(\mathbb{W}, \mathbb{S}) \xmapsto{\mathbb{T}}{}^{+} \mathbf{abort}$.
Proof: From $(W, \mathcal{S}) \preceq (\mathbb{W}, \mathbb{S}) \diamond (\mathscr{M}, \zeta, \mathscr{L})$, 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 $(\mathbb{W}, \mathbb{S}) \xmapsto{\mathbb{T}}{}^{+} \mathbf{abort}$.

(4) If $(W, \mathcal{S}) \xmapsto{\iota} (W', \mathcal{S}')$ and $\chi = \iota :: \chi'$, then
there exist t, $\mathbb{T}$, $\mathbb{W}'$, $\mathbb{S}'$ and $\mathcal{M}'$ such that all the following hold:
(a) $(\mathbb{W}, \mathbb{S}) \xmapsto{\mathbb{T}}{}^{*} (\mathbb{W}', \mathbb{S}')$;
(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) $\chi' \models (W', \mathcal{S}') \preceq (\mathbb{W}', \mathbb{S}') \diamond \mathcal{M}'$;
(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 $(W, \mathcal{S}) \preceq (\mathbb{W}, \mathbb{S}) \diamond (\mathscr{M}, \zeta, \mathscr{L})$, we know there exist t, $\mathbb{T}$, $\mathbb{W}'$, $\mathbb{S}'$, $\mathscr{M}'$ and ζ' such that all the following hold:
(A) $(\mathbb{W}, \mathbb{S}) \xmapsto{\mathbb{T}}{}^{*} (\mathbb{W}', \mathbb{S}')$;
(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) $(W', \mathcal{S}') \preceq (\mathbb{W}', \mathbb{S}') \diamond (\mathscr{M}', \zeta', \mathscr{L})$;
(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 $\zeta(\mathsf{t}) \subseteq \zeta'(\mathsf{t})$ and for any $\mathsf{t}' \in \{\mathsf{t}' \mid \mathsf{dep}(\mathsf{t}, \zeta, \mathsf{t}')\}$, $\zeta(\mathsf{t}') \subseteq \zeta'(\mathsf{t}')$;
(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 $\zeta(\mathsf{t}') \subseteq \zeta'(\mathsf{t}')$ and for any $\mathsf{t}'' \in \{\mathsf{t}'' \mid \mathsf{dep}(\mathsf{t}', \zeta, \mathsf{t}'')\}$, $\zeta(\mathsf{t}'') \subseteq \zeta'(\mathsf{t}'')$ and $\neg\mathsf{dep}(\mathsf{t}', \zeta, \mathsf{t})$
or $\mathscr{M}'(\mathsf{t}') = \mathscr{M}(\mathsf{t}')$ and $\zeta(\mathsf{t}') \subseteq \zeta'(\mathsf{t}')$ and for any $\mathsf{t}'' \in \{\mathsf{t}'' \mid \mathsf{dep}(\mathsf{t}', \zeta, \mathsf{t}'')\}$, $\zeta(\mathsf{t}'') \subseteq \zeta'(\mathsf{t}'')$ and $\mathsf{dep}(\mathsf{t}', \zeta, \mathsf{t})$ and $\zeta(t) \neq \emptyset$.
Since $\chi \models O^{\mathsf{co}}_{f\omega}(W, \mathcal{S}, \chi_o)$, by Lemmas B.28 and B.29, we know there exists χ'_o such that $\chi_o = (\mathsf{get\_obsv}(\iota)) :: \chi'_o$ and $\chi' \models O^{\mathsf{co}}_{f\omega}(W', \mathcal{S}', \chi'_o)$.
By (B.6), we know there exists χ'_x such that $\mathsf{roundsub}(\chi', \mathsf{activeThrds}(W'), \chi'_x)$.
Choose $\mathcal{M}'$ such that $dom(\mathcal{M}') = \mathsf{activeThrds}(W')$ and for any $\mathsf{t} \in dom(\mathcal{M}')$, let $\xi'_n = \{\mathsf{t}'_n \mid \mathsf{nonblocker}(\mathsf{t}, \mathsf{t}'_n, \zeta')\}$, we have
$\mathcal{M}'(\mathsf{t}) = (\mathscr{M}'(\mathsf{t}), (\{\mathsf{t}' \rightsquigarrow \zeta'(\mathsf{t}') \mid \mathsf{dep}(\mathsf{t}, \zeta', \mathsf{t}') \vee \mathsf{t} = \mathsf{t}'\}, dom(\zeta')), \mathsf{minPos}(\chi'_x, \xi'_n))$.

By the co-induction hypothesis, we know $\chi' \models (W', \mathcal{S}') \preceq (\mathbb{W}', \mathbb{S}') \diamond \mathcal{M}'$.
Below we prove: either $\mathrm{t} \in \mathrm{tidset}(\mathbb{T})$, or $\mathcal{M}'(\mathrm{t}) < \mathcal{M}(\mathrm{t})$. From (D), if $\mathrm{t} \in \mathrm{tidset}(\mathbb{T})$, we are done. If $\mathscr{M}'(\mathrm{t}) < \mathscr{M}(\mathrm{t})$, then $\mathcal{M}'(\mathrm{t}) < \mathcal{M}(\mathrm{t})$.

- If $\mathscr{M}'(\mathrm{t}) = \mathscr{M}(\mathrm{t})$ and $\zeta(\mathrm{t}) \neq \emptyset$ and $\zeta(\mathrm{t}) \subseteq \zeta'(\mathrm{t})$ and there exists t' such that $\mathrm{t}' \in \{\mathrm{t}' \mid \mathrm{dep}(\mathrm{t}, \zeta, \mathrm{t}') \vee \mathrm{t} = \mathrm{t}'\}$, $\zeta(\mathrm{t}') \subset \zeta'(\mathrm{t}')$.
  From the premise, we know
  $$dom(\zeta') = \mathrm{activeThrds}(W'), \quad dom(\zeta) = \mathrm{activeThrds}(W)$$
  By the operational semantics, we know $\mathrm{activeThrds}(W') \subseteq \mathrm{activeThrds}(W)$. And for any $\mathrm{t}'' \in \{\mathrm{t}' \mid \mathrm{dep}(\mathrm{t}, \zeta, \mathrm{t}')\}$, we know $\zeta(\mathrm{t}'') \subseteq \zeta'(\mathrm{t}'')$, $\zeta(\mathrm{t}'') \subseteq dom(\zeta)$ and $\zeta'(\mathrm{t}'') \subseteq dom(\zeta')$.
  Thus we have $(\{\mathrm{t}' \rightsquigarrow \zeta(\mathrm{t}') \mid \mathrm{dep}(\mathrm{t}, \zeta, \mathrm{t}') \vee \mathrm{t} = \mathrm{t}'\}, dom(\zeta)) < (\{\mathrm{t}' \rightsquigarrow \zeta'(\mathrm{t}') \mid \mathrm{dep}(\mathrm{t}, \zeta', \mathrm{t}') \vee \mathrm{t} = \mathrm{t}'\}, dom(\zeta'))$, then $\mathcal{M}'(\mathrm{t}) < \mathcal{M}(\mathrm{t})$
- If $\mathscr{M}'(\mathrm{t}) = \mathscr{M}(\mathrm{t})$ and $\zeta(\mathrm{t}) \neq \emptyset$ and $\zeta(\mathrm{t}) \subseteq \zeta'(\mathrm{t})$ and for any t' such that $\mathrm{t}' \in \{\mathrm{t}' \mid \mathrm{dep}(\mathrm{t}, \zeta, \mathrm{t}') \vee \mathrm{t} = \mathrm{t}'\}$, $\zeta(\mathrm{t}') = \zeta'(\mathrm{t}')$. We have $(\{\mathrm{t}' \rightsquigarrow \zeta(\mathrm{t}') \mid \mathrm{dep}(\mathrm{t}, \zeta, \mathrm{t}') \vee \mathrm{t} = \mathrm{t}'\}, dom(\zeta)) = (\{\mathrm{t}' \rightsquigarrow \zeta'(\mathrm{t}') \mid \mathrm{dep}(\mathrm{t}, \zeta', \mathrm{t}') \vee \mathrm{t} = \mathrm{t}'\}, dom(\zeta'))$
  Let $\xi_n = \{\mathrm{t}_n \mid \mathrm{nonblocker}(\mathrm{t}, \mathrm{t}_n, \zeta)\}$. We can prove $\xi_n = \xi'_n$
  Below we prove $\mathrm{minPos}(\chi'_x, \xi_n) < \mathrm{minPos}(\chi_x, \xi_n)$
  Since $\mathrm{roundsub}(\chi, \mathrm{activeThrds}(W), \chi_x)$, we know there exists χ_z such that $\chi = \chi_x :: \chi_z$. Since $\chi \neq \epsilon$ and $\mathrm{activeThrds}(W) \neq \emptyset$, we know $\chi_x \neq \epsilon$. Since $\chi = \iota :: \chi'$, we know there exists χ_y such that $\chi_x = \iota :: \chi_y$ and $\chi' = \chi_y :: \chi_z$. Since $\zeta(\mathrm{t}) \neq \emptyset$, so $\mathrm{t} \notin \xi_n$, and $\mathrm{t} = \mathrm{tid}(\lfloor \iota \rfloor)$, we know
  $$\mathrm{minPos}(\chi_y, \xi_n) < \mathrm{minPos}(\chi_x, \xi_n).$$
  By B.8 we have $\mathrm{roundsub}(\chi', \xi_n, \chi_y)$. Since $\xi_n = \xi'_n \subseteq \mathrm{activeThrds}(W')$, from $\mathrm{roundsub}(\chi', \mathrm{activeThrds}(W'), \chi'_x)$, we have $\mathrm{roundsub}(\chi', \xi_n, \chi'_x)$. Then by Lemma B.30, we know
  $$\mathrm{minPos}(\chi_y, \xi_n) = \mathrm{minPos}(\chi'_x, \xi_n).$$
  Thus $\mathcal{M}'(\mathrm{t}) < \mathcal{M}(\mathrm{t})$.

Next we prove: for any $\mathrm{t}' \in dom(\mathcal{M}) \backslash \{\mathrm{t}\}$, either $\mathrm{t}' \in \mathrm{tidset}(\mathbb{T})$, or $\mathcal{M}'(\mathrm{t}') < \mathcal{M}(\mathrm{t}')$. From (E), if $\mathrm{t}' \in \mathrm{tidset}(\mathbb{T})$, we are done. If $\mathscr{M}'(\mathrm{t}') < \mathscr{M}(\mathrm{t}')$, then $\mathcal{M}'(\mathrm{t}) < \mathcal{M}(\mathrm{t})$.

- If $\mathscr{M}'(\mathrm{t}') = \mathscr{M}(\mathrm{t}')$ and $\zeta(\mathrm{t}') \subseteq \zeta'(\mathrm{t}')$ and there exists t'' such that $\mathrm{t}'' \in \{\mathrm{t}'' \mid \mathrm{dep}(\mathrm{t}', \zeta, \mathrm{t}'') \vee \mathrm{t}' = \mathrm{t}''\}$, $\zeta(\mathrm{t}'') \subset \zeta'(\mathrm{t}'')$.
  From the premise, we know
  $$dom(\zeta') = \mathrm{activeThrds}(W'), \quad dom(\zeta) = \mathrm{activeThrds}(W)$$
  By the operational semantics, we know $\mathrm{activeThrds}(W') \subseteq \mathrm{activeThrds}(W)$.
  And for any $\mathrm{t}''' \in \{\mathrm{t}''' \mid \mathrm{dep}(\mathrm{t}', \zeta, \mathrm{t}''')\}$, we know $\zeta(\mathrm{t}''') \subseteq \zeta'(\mathrm{t}''')$, $\zeta(\mathrm{t}''') \subseteq dom(\zeta)$ and $\zeta'(\mathrm{t}''') \subseteq dom(\zeta')$.
  Thus we have $(\{\mathrm{t} \rightsquigarrow \zeta(\mathrm{t}) \mid \mathrm{dep}(\mathrm{t}', \zeta, \mathrm{t}) \vee \mathrm{t} = \mathrm{t}'\}, dom(\zeta)) < (\{\mathrm{t} \rightsquigarrow \zeta'(\mathrm{t}) \mid \mathrm{dep}(\mathrm{t}', \zeta', \mathrm{t}) \vee \mathrm{t} = \mathrm{t}'\}, dom(\zeta'))$, then $\mathcal{M}'(\mathrm{t}') < \mathcal{M}(\mathrm{t}')$
- If $\mathscr{M}'(\mathrm{t}') = \mathscr{M}(\mathrm{t}')$ and $\zeta(\mathrm{t}') \subseteq \zeta'(\mathrm{t}')$ and for any t'' such that $\mathrm{t}'' \in \{\mathrm{t}'' \mid \mathrm{dep}(\mathrm{t}', \zeta, \mathrm{t}'') \vee \mathrm{t}' = \mathrm{t}''\}$, $\zeta(\mathrm{t}'') = \zeta'(\mathrm{t}'')$. We have $(\{\mathrm{t} \rightsquigarrow \zeta(\mathrm{t}) \mid \mathrm{dep}(\mathrm{t}', \zeta, \mathrm{t}) \vee \mathrm{t} = \mathrm{t}'\}, dom(\zeta)) = (\{\mathrm{t} \rightsquigarrow \zeta'(\mathrm{t}) \mid \mathrm{dep}(\mathrm{t}', \zeta', \mathrm{t}) \vee \mathrm{t} = \mathrm{t}'\}, dom(\zeta'))$
  Let $\xi_n = \{\mathrm{t}_n \mid \mathrm{nonblocker}(\mathrm{t}', \mathrm{t}_n, \zeta)\}$. We can prove $\xi_n = \xi'_n$
  Below we prove $\mathrm{minPos}(\chi'_x, \xi_n) < \mathrm{minPos}(\chi_x, \xi_n)$
  Since $\mathrm{roundsub}(\chi, \mathrm{activeThrds}(W), \chi_x)$, we know there exists χ_z such that $\chi = \chi_x :: \chi_z$. Since $\chi \neq \epsilon$ and $\mathrm{activeThrds}(W) \neq \emptyset$, we know $\chi_x \neq \epsilon$. Since $\chi = \iota :: \chi'$, we know there exists χ_y such that $\chi_x = \iota :: \chi_y$ and $\chi' = \chi_y :: \chi_z$. From $(\neg \mathrm{dep}(\mathrm{t}', \zeta, \mathrm{t}) or (\mathrm{dep}(\mathrm{t}', \zeta, \mathrm{t}) and \zeta(\mathrm{t}) \neq \emptyset))$, we know $\mathrm{t} \notin \xi_n$, and $\mathrm{t} = \mathrm{tid}(\lfloor \iota \rfloor)$, we know
  $$\mathrm{minPos}(\chi_y, \xi_n) < \mathrm{minPos}(\chi_x, \xi_n).$$

By B.8 we have $\mathsf{roundsub}(\chi', \xi_n, \chi_y)$. Since $\xi_n = \xi'_n \subseteq \mathsf{activeThrds}(W')$, from $\mathsf{roundsub}(\chi', \mathsf{activeThrds}(W'), \chi'_x)$, we have $\mathsf{roundsub}(\chi', \xi_n, \chi'_x)$. Then by Lemma B.30, we know

$$\mathsf{minPos}(\chi_y, \xi_n) = \mathsf{minPos}(\chi'_x, \xi_n).$$

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

Thus we are done. □

Lemma B.28. *If* $(\iota :: \chi_x) \models O^{\mathrm{co}}_{f\omega}(W, \mathcal{S}, \chi_a)$, *then there exists* χ_b *such that* $\chi_a = (\mathsf{get\_obsv}(\iota)) :: \chi_b$.

Lemma B.29. *If* $(\iota :: \chi_x) \models O^{\mathrm{co}}_{f\omega}(W, \mathcal{S}, \chi_a)$, $(W, \mathcal{S}) \xrightarrow{\iota} (W_x, \mathcal{S}_x)$ *and* $\chi_a = (\mathsf{get\_obsv}(\iota)) :: \chi_b$, *then* $\chi_x \models O^{\mathrm{co}}_{f\omega}(W_x, \mathcal{S}_x, \chi_b)$.

Proof. By co-induction, and then by inversion three times over $(e :: \chi_x) \models O^{\mathrm{co}}_{f\omega}(W, \mathcal{S}, \chi_a)$. □

Lemma B.30. *If* $\xi \neq \emptyset$, $\mathsf{roundsub}(\chi, \xi, \chi_x)$ *and* $\mathsf{roundsub}(\chi, \xi, \chi_y)$, *then* $\mathsf{minPos}(\chi_x, \xi) = \mathsf{minPos}(\chi_y, \xi)$.

Proof. From the premises, we know $|\chi_x| \neq \omega$ and $|\chi_y| \neq \omega$. Suppose $|\chi_x| \leq |\chi_y|$. We have two cases:

- $|\chi| \neq \omega$ and $\chi_x = \chi$:
  We know there exists χ'_y such that $\chi = \chi_y :: \chi'_y$. Thus $\chi_y = \chi$. Then we have $\chi_x = \chi_y$ and we are done.
- there exists χ'_x such that $\chi = \chi_x :: \chi'_x$ and $\xi \subseteq \mathsf{tidset}(\chi_x)$:
  We know there exists χ'_y such that $\chi = \chi_y :: \chi'_y$. Thus there exists χ_z such that $\chi_y = \chi_x :: \chi_z$ and $\chi'_x = \chi_z :: \chi'_y$. Since $\xi \neq \emptyset$, we know $\mathsf{tidset}(\chi_x) \neq \emptyset$. Thus there exist e and χ_0 such that $\chi_x = e :: \chi_0$. Thus $\chi_y = e :: \chi_0 :: \chi_z$. By induction over $|\chi_0|$.

□

Proof of Lemma B.27. 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 28 defines F and gfp F (i.e., $O^{\text{m}}_{f\!o}$ at the bottom part of the figure).

$$S \stackrel{\text{def}}{=} \{(\mathcal{M}, \mathbb{W}, \mathbb{S}, \chi_o) \mid \exists \chi, W, \mathcal{S}.\ (\chi \models O^{\text{co}}_{f\!o}(W, \mathcal{S}, \chi_o)) \wedge (\chi \models (W, \mathcal{S}) \preceq (\mathbb{W}, \mathbb{S}) \diamond \mathcal{M})\}.$$

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

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

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

(1) $(\chi \models (W, \mathcal{S}) \preceq (\mathbb{W}, \mathbb{S}) \diamond \mathcal{M})$, $n > 0$, $(W, \mathcal{S}) \stackrel{\chi}{\longmapsto}{}^{n}$ **abort** and $\text{get\_obsv}(\chi) = \chi_o$:

(2) $(\chi \models (W, \mathcal{S}) \preceq (\mathbb{W}, \mathbb{S}) \diamond \mathcal{M})$, $n > 0$, $(W, \mathcal{S}) \stackrel{\chi}{\longmapsto}{}^{n}$ (**skip**, _) and $\text{get\_obsv}(\chi) = \chi_o$:

(3) $(\chi \models (W, \mathcal{S}) \preceq (\mathbb{W}, \mathbb{S}) \diamond \mathcal{M})$, $n > 0$, $\chi = \chi_x :: \chi_y$, $\chi_o = \epsilon$, $(W, \mathcal{S}) \stackrel{\chi_x}{\longmapsto}{}^{n} (W_y, \mathcal{S}_y)$, $\text{get\_obsv}(\chi_x) = \epsilon$, $(\chi_y \models O^{\text{co}}_{f\!o}(W_y, \mathcal{S}_y, \epsilon))$, $\text{activeThrds}(W) \subseteq \text{tidset}(\chi_x)$:
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

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

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

$$\begin{array}{l}
\forall n, \chi, W, \mathcal{S}, \mathbb{W}, \mathbb{S}, \mathcal{M}, \chi_x, \chi_y, W_y, \mathcal{S}_y, \chi_o.\\
(n > 0) \wedge (\chi \models (W, \mathcal{S}) \preceq (\mathbb{W}, \mathbb{S}) \diamond \mathcal{M}) \wedge (\chi = \chi_x :: \chi_y)\\
\wedge ((W, \mathcal{S}) \stackrel{\chi_x}{\longmapsto}{}^{n} (W_y, \mathcal{S}_y)) \wedge (\text{get_obsv}(\chi_x) = \chi_o)\\
\implies\\
\exists \mathbb{T}_x, \mathbb{W}_y, \mathbb{S}_y, \mathcal{M}_y.\\
((\mathbb{W}, \mathbb{S}) \stackrel{\mathbb{T}_x}{\longmapsto}{}^{*} (\mathbb{W}_y, \mathbb{S}_y)) \wedge (\text{get_obsv}(\mathbb{T}_x) = \chi_o)\\
\wedge (\chi_y \models (W_y, \mathcal{S}_y) \preceq (\mathbb{W}_y, \mathbb{S}_y) \diamond \mathcal{M}_y)\\
\wedge (dom(\mathcal{M}) = \text{activeThrds}(\mathbb{W}) \neq \emptyset) \wedge (\forall \mathrm{t} \in dom(\mathcal{M}) \backslash \text{tidset}(\mathbb{T}_x).\ \mathcal{M}_y(\mathrm{t}) < \mathcal{M}(\mathrm{t}))
\end{array} \tag{B.9}$$

By induction over n.

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

(D) for any $\mathrm{t}' \in dom(\mathcal{M})$, either $\mathrm{t}' \in \mathsf{tidset}(\mathbb{T})$, or $\mathcal{M}'(\mathrm{t}') < \mathcal{M}(\mathrm{t}')$.
From (D), we have: $\forall \mathrm{t} \in dom(\mathcal{M}) \backslash \mathsf{tidset}(\mathbb{T}).\ \mathcal{M}'(\mathrm{t}) < \mathcal{M}(\mathrm{t})$.
From the induction hypothesis, we have:

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

Thus we have $(\mathbb{W}, \mathbb{S}) \overset{\mathbb{T}::\mathbb{T}_x}{\longmapsto}{}^* (\mathbb{W}_y, \mathbb{S}_y)$ and $\mathsf{get\_obsv}(\mathbb{T} :: \mathbb{T}_x) = \mathsf{get\_obsv}(e :: \chi'_x) = \mathsf{get\_obsv}(\chi_x)$. Also, for any $\mathrm{t} \in dom(\mathcal{M}) \backslash \mathsf{tidset}(\mathbb{T} :: \mathbb{T}_x)$, we know $\mathrm{t} \in dom(\mathcal{M}) \backslash \mathsf{tidset}(\mathbb{T}_x)$. Thus $\mathcal{M}'(\mathrm{t}) < \mathcal{M}(\mathrm{t})$. Also, we know $\mathrm{t} \in dom(\mathcal{M}') \backslash \mathsf{tidset}(\mathbb{T}_x)$. Thus $\mathcal{M}_y(\mathrm{t}) < \mathcal{M}'(\mathrm{t})$. By the transitivity of the well-founded order, we have $\mathcal{M}_y(\mathrm{t}) < \mathcal{M}(\mathrm{t})$.

(4) $(\chi \models (W, \mathcal{S}) \preceq (\mathbb{W}, \mathbb{S}) \diamond \mathcal{M})$, $n > 0$, $\chi = \chi_x :: \chi_y$, $\chi_o = e_a :: \chi_a :: \chi_b$, $(W, \mathcal{S}) \overset{\chi_x}{\longmapsto}{}^n (W_y, \mathcal{S}_y)$, $\mathsf{get\_obsv}(\chi_x) = e_a :: \chi_a$, $(\chi_y \models O^{\mathrm{co}}_{f\omega}(W_y, \mathcal{S}_y, \chi_b))$, $\mathsf{activeThrds}(W) \subseteq \mathsf{tidset}(\chi_x)$:
To prove $(\mathcal{M}, \mathbb{W}, \mathbb{S}, \chi_o) \in F(S)$, we want to prove: there exist $\mathbb{T}_x$, $\mathbb{W}_y$, $\mathbb{S}_y$ and $\mathcal{M}_y$ such that

$$\begin{array}{c}(\mathbb{W}, \mathbb{S}) \overset{\mathbb{T}_x}{\longmapsto}{}^* (\mathbb{W}_y, \mathbb{S}_y), \quad \mathsf{get\_obsv}(\mathbb{T}_x) = e_a :: \chi_a, \quad (\mathcal{M}_y, \mathbb{W}_y, \mathbb{S}_y, \chi_b) \in S, \\ dom(\mathcal{M}) = \mathsf{activeThrds}(\mathbb{W}) \neq \emptyset, \quad \forall \mathrm{t} \in dom(\mathcal{M}) \backslash \mathsf{tidset}(\mathbb{T}_x).\ \mathcal{M}_y(\mathrm{t}) < \mathcal{M}(\mathrm{t})\ .\end{array}$$

Since $(\chi_y \models O^{\mathrm{co}}_{f\omega}(W_y, \mathcal{S}_y, \chi_b))$, we are done by (B.9).

□

B.6 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 [Filipović et al. 2010; Liang and Feng 2013; Liang et al. 2013], we prove $\Pi \preceq_P^{\mathsf{lin}} \Gamma$ is equivalent to $\Pi \sqsubseteq_P^{\mathsf{fin}} \Gamma$. This equivalence result (Theorem B.38) 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 B.39 and B.40) between linearizability, starvation-freedom, deadlock-freedom and the contextual refinement under fair scheduling.

Formalizing starvation-freedom and deadlock-freedom. Following the earlier work [Herlihy and Shavit 2011; Liang et al. 2013], we define starvation-freedom and deadlock-freedom over execution traces. We first introduce some auxiliary definitions in Fig. 29. 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 $(\lfloor W \rfloor, \mathcal{S}) \xrightarrow{\mathcal{E}}{}^* (W', \mathcal{S}')$ represents a zero or multi-step execution of $(\lfloor W \rfloor, \mathcal{S})$ that leads to $(W', \mathcal{S}')$ with the sequence $\mathcal{E}$ of events generated; $(\lfloor W \rfloor, \mathcal{S}) \xrightarrow{\mathcal{E}}{}^\omega \cdot$ then represents an infinite execution with the *whole* event trace $\mathcal{E}$, which must be infinite too. We also insert a $(\mathbf{spawn}, n)$ 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 $(\mathsf{t}, \mathbf{term})$ event to mark the termination of this thread t.

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

Also in Fig. 29, we define $\mathsf{prog\text{-}t}(\mathcal{E})$ to say that every method call in $\mathcal{E}$ eventually finishes. Here $\mathsf{pend\_inv}(\mathcal{E}(1..i))$ gets the set of pending invocation events in the sub-trace $\mathcal{E}(1), \ldots, \mathcal{E}(i)$ of $\mathcal{E}$, and $\mathsf{match}(e, \mathcal{E}(j))$ says that $\mathcal{E}(j)$ is a return event which matches with the invocation e, so the corresponding method call finishes. $\mathsf{prog\text{-}p}(\mathcal{E})$ requires that *some* pending method call finishes. Different from $\mathsf{prog\text{-}t}(\mathcal{E})$, the return event $\mathcal{E}(j)$ in $\mathsf{prog\text{-}p}(\mathcal{E})$ does not have to be a matching return of the pending invocation e. We also write $\mathsf{abt}(\mathcal{E})$ to say that $\mathcal{E}$ ends with a fault event.

Starvation-freedom and deadlock-freedom require prog-t and prog-p, respectively, in every fair execution. They are defined below.

Definition B.31 (Starvation-free objects). $\mathsf{starvation\text{-}free}_P(\Pi)$ iff

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

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

Definition B.32 (Deadlock-free objects). $\mathsf{deadlock\text{-}free}_P(\Pi)$ iff

$\forall n, C_1, \ldots, C_n, \sigma_c, \sigma_o, \mathcal{E}.\ \mathcal{E} \in \mathcal{T}_\omega[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \parallel \ldots \parallel C_n), (\sigma_c, \sigma_o, \circledcirc)]\!] \wedge ((\sigma_o, \_) \models P) \wedge \mathsf{fair}(\mathcal{E})$
$\implies \mathsf{prog\text{-}p}(\mathcal{E}) \vee \mathsf{abt}(\mathcal{E})$.

Formalizing linearizability. Linearizability describes atomic behaviors of object implementations. Following its standard definition [Herlihy and Wing 1990], we define linearizability using histories, which are finite event traces $\mathcal{E}$ consisting of only method invocations, returns, and object faults. As defined in Fig. 29, 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 $\mathcal{E}$ if no matching return follows it. We use $\mathsf{pend\_inv}(\mathcal{E})$ to get the set of pending invocations in $\mathcal{E}$. We complete a history $\mathcal{E}$ by appending zero or more return events, and dropping the remaining pending invocations. The result is denoted by

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

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

$$|\mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \parallel \ldots \parallel C_n| \stackrel{\text{def}}{=} n \qquad \mathsf{tnum}((\mathbf{spawn},n)::\mathcal{E}) \stackrel{\text{def}}{=} n$$

$$\mathsf{fair}(\mathcal{E}) \ \text{ iff } \ |\mathcal{E}| = \omega \implies \forall \mathrm{t} \in [1..\mathsf{tnum}(\mathcal{E})].\ |(\mathcal{E}|_{\mathrm{t}})| = \omega \vee \mathsf{last}(\mathcal{E}|_{\mathrm{t}}) = (\mathrm{t},\mathbf{term})$$

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

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

$$\text{prog-t}(\mathcal{E}) \ \text{ iff } \ \forall i,e.\ e \in \mathsf{pend\_inv}(\mathcal{E}(1..i)) \implies \exists j.\ j > i \wedge \mathsf{match}(e,\mathcal{E}(j))$$

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

$$\mathsf{abt}(\mathcal{E}) \ \text{ iff } \ \exists i.\ \mathsf{is\_abt}(\mathcal{E}(i))$$

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

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

$$\mathcal{T}[\![W,\mathcal{S}]\!] \stackrel{\text{def}}{=} \{\mathcal{E} \mid \exists W',\mathcal{S}'.\ (W,\mathcal{S}) \overset{\mathcal{E}}{\longmapsto}^{*} (W',\mathcal{S}') \vee (W,\mathcal{S}) \overset{\mathcal{E}}{\longmapsto}^{*} \mathbf{abort}\}$$

$$\mathcal{H}[\![W,\mathcal{S}]\!] \stackrel{\text{def}}{=} \{\mathsf{get\_hist}(\mathcal{E}) \mid \mathcal{E} \in \mathcal{T}[\![W,\mathcal{S}]\!]\}$$

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

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

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

Fig. 29. Event traces.

completions($\mathcal{E}$). It is a set of histories, and for each history in it, every invocation has a matching return event.

Definition B.33 (Linearizable histories). $\mathcal{E} \preceq^{\mathsf{lin}} \mathcal{E}'$ iff both the following hold.

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

That is, $\mathcal{E}$ is linearizable with respect to $\mathcal{E}'$ 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. 29, we use $\Gamma \rhd (\Sigma,\mathcal{E}')$ to mean that $\mathcal{E}'$ is a legal sequential history generated by any client using the specification Γ with an abstract initial state Σ.

Definition B.34 (Extensions of a history). extensions($\mathcal{E}$) is a set of well-formed histories where we extend $\mathcal{E}$ by appending return events. It is inductively defined as follows.

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

Definition B.35 (Completions of a history). $\mathsf{truncate}(\mathcal{E})$ is the maximal complete sub-history of $\mathcal{E}$. It is inductively defined by dropping the pending invocations in $\mathcal{E}$.

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

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

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

Definition B.36 (Linearizability of objects). The object implementation Π is linearizable with respect to Γ, written as $\Pi \preceq_P^{\mathsf{lin}} \Gamma$, iff

$$\forall n, C_1, \ldots, C_n, \sigma_c, \sigma, \Sigma, \mathcal{E}.\ \mathcal{E} \in \mathcal{H}[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma, \circledcirc)]\!] \wedge ((\sigma, \Sigma) \models P)$$
$$\Longrightarrow \exists \mathcal{E}_c, \mathcal{E}'.\ \mathcal{E}_c \in \mathsf{completions}(\mathcal{E}) \wedge \Gamma \rhd (\Sigma, \mathcal{E}') \wedge \mathcal{E}_c \preceq^{\mathsf{lin}} \mathcal{E}'$$

Equivalence results.

Definition B.37 (Basic contextual refinement). $\Pi \sqsubseteq_P^{\mathsf{fin}} \Pi'$ iff

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

Theorem B.38 (Basic equivalence for linearizability). $\Pi \preceq_P^{\mathsf{lin}} \Gamma \iff \Pi \sqsubseteq_P^{\mathsf{fin}} \Gamma$.

Theorem B.39 (① in Fig. 25). $\Pi \preceq_P^{\mathsf{lin}} \Gamma \wedge \mathsf{starvation\text{-}free}_P(\Pi) \iff \Pi \sqsubseteq_P \Gamma$.

Theorem B.40 (② in Fig. 25). $\Pi \preceq_P^{\mathsf{lin}} \Gamma \wedge \mathsf{deadlock\text{-}free}_P(\Pi) \iff \Pi \sqsubseteq_{\mathsf{wr}_1(P)} \mathsf{wr}_1(\Gamma)$.

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

B.6.1 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. $x := \mathbf{rand}(m)$ assigns x a random integer $i \in [1..m]$, and $x := \mathbf{rand}()$ computes an arbitrarily large random integer. Then, for any n, we define MGC_n as follows.

$$\begin{array}{rcl} \mathsf{MGT}_\mathsf{t} & \stackrel{\text{def}}{=} & \mathbf{while}\ (\mathbf{true})\{ \\ & & \quad x_\mathsf{t} := \mathbf{rand}();\ y_\mathsf{t} := \mathbf{rand}(m); \\ & & \quad z_\mathsf{t} := f_{y_\mathsf{t}}(x_\mathsf{t}); \\ & & \} \\ \mathsf{MGC}_n & \stackrel{\text{def}}{=} & \|_{\mathsf{t} \in [1..n]}\, \mathsf{MGT}_\mathsf{t} \end{array}$$

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.

$$\begin{array}{rcl}
\mathsf{MGTp}_t & \stackrel{\text{def}}{=} & \textbf{while}\ (\textbf{true})\{ \\
& & \quad x_t := \textbf{rand}();\ y_t := \textbf{rand}(m);\ \textbf{print}(y_t, x_t); \\
& & \quad z_t := f_{y_t}(x_t);\ \textbf{print}(z_t); \\
& & \} \\
\mathsf{MGCp}_n & \stackrel{\text{def}}{=} & \|_{t\in[1..n]}\, \mathsf{MGTp}_t
\end{array}$$

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.

$$\begin{array}{rcl}
\mathsf{MGTp1}_t & \stackrel{\text{def}}{=} & \textbf{while}\ (\ \textbf{rand}() > 0\ )\{ \\
& & \quad x_t := \textbf{rand}();\ y_t := \textbf{rand}(m);\ \textbf{print}(0); \\
& & \quad z_t := f_{y_t}(x_t);\ \textbf{print}(1); \\
& & \} \\
& & \textbf{print}(2); \\
\mathsf{MGCp1}_n & \stackrel{\text{def}}{=} & \|_{t\in[1..n]}\, \mathsf{MGTp1}_t
\end{array}$$

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

$$\sigma_{\mathsf{MGC}(n)} \stackrel{\text{def}}{=} \{x_t \rightsquigarrow \_, y_t \rightsquigarrow \_, z_t \rightsquigarrow \_ \mid 1 \le t \le n\}$$

B.6.2 Basic equivalence for linearizability. Follow earlier work [Liang et al. 2013], we first define the MGC versions of "linearizability" and "refinement".

Definition B.41. $\Pi \preceq_P^{\mathsf{MGC}} \Gamma$ iff

$$\begin{array}{l}
\forall n, \sigma_{\mathsf{MGC}(n)}, \sigma, \Sigma, \mathcal{E}.\ \mathcal{E} \in \mathcal{H}[\![(\textbf{let}\ \Pi\ \textbf{in}\ \mathsf{MGC}_n), (\sigma_{\mathsf{MGC}(n)}, \sigma, \circledcirc)]\!] \wedge ((\sigma, \Sigma) \models P) \\
\Longrightarrow \exists \mathcal{E}_c, \mathcal{E}'.\ \mathcal{E}_c \in \mathsf{completions}(\mathcal{E}) \wedge \Gamma \rhd_n (\sigma_{\mathsf{MGC}(n)}, \Sigma, \mathcal{E}') \wedge \mathcal{E}_c \preceq^{\mathsf{lin}} \mathcal{E}'
\end{array}$$

where

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

Definition B.42. $\Pi \subseteq_P \Gamma$ iff

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

To prove Theorem B.38, we prove the following lemmas.

Lemma B.43. $\Pi \preceq_P^{\mathsf{lin}} \Gamma \iff \Pi \preceq_P^{\mathsf{MGC}} \Gamma$.

Lemma B.44. $\Pi \sqsubseteq_P^{\mathsf{fin}} \Gamma \iff \Pi \subseteq_P \Gamma$.

Lemma B.45. $\Pi \subseteq_P \Gamma \iff \Pi \preceq_P^{\mathsf{MGC}} \Gamma$.

Proof of Lemma B.43. 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 B.36) 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 B.46 (MGC is the most general). *For any $n, C_1, \ldots, C_n, \sigma_c, \sigma_{\mathsf{MGC}(n)}$ and σ_o, we have $\mathcal{H}[\![(\textbf{let}\ \Pi\ \textbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_o, \circledcirc)]\!] \subseteq \mathcal{H}[\![(\textbf{let}\ \Pi\ \textbf{in}\ \mathsf{MGC}_n), (\sigma_{\mathsf{MGC}(n)}, \sigma_o, \circledcirc)]\!]$.*

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.10).

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 $(W_1, \mathcal{S}_1) \overset{e_1}{\longmapsto}$ **abort** and is_obj_abt(e_1), then
there exists $\mathcal{E}_2$ such that $(W_2, \mathcal{S}_2) \overset{\mathcal{E}_2}{\longmapsto}{}^*$ **abort** and
$e_1 = \mathsf{get\_hist}(\mathcal{E}_2)$;
(2) if $(W_1, \mathcal{S}_1) \overset{e_1}{\longmapsto} (W_1', \mathcal{S}_1')$, then
there exist $\mathcal{E}_2$, W_2' and $\mathcal{S}_2'$ such that $(W_2, \mathcal{S}_2) \overset{\mathcal{E}_2}{\longmapsto}{}^* (W_2', \mathcal{S}_2')$,
$\mathsf{get\_hist}(e_1) = \mathsf{get\_hist}(\mathcal{E}_2)$ and $(W_1', \mathcal{S}_1') \lesssim_{\mathsf{MGC}} (W_2', \mathcal{S}_2')$.

(B.10)

The simulation relation is constructed as shown in Fig. 30(a). That is, for each client thread t, if the left side is in some normal client code, it corresponds to MGT_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 MGT_t.
(2) Each method invocation corresponds to the steps executing MGT_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.10), 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 B.43 is immediate by unfolding the definitions of $\Pi \preceq_P^{\mathsf{MGC}} \Gamma$ and $\Pi \preceq_P^{\mathsf{lin}} \Gamma$, and applying Lemma B.46.

Proof of Lemma B.44.

(1) $\Pi \sqsubseteq_P^{\mathsf{fin}} \Gamma \implies \Pi \subseteq_P \Gamma$:
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 B.47. *For any n, σ_o and $\sigma_{\mathsf{MGC}(n)}$, both the following holds.*
(a) For any $\mathcal{E}_1$, if $\mathcal{E}_1 \in \mathcal{H}[\![(\textbf{let } \Pi \textbf{ in } \mathsf{MGC}_n), (\sigma_{\mathsf{MGC}(n)}, \sigma_o, \circledcirc)]\!]$, then there exists $\mathcal{E}_2$ such that $\mathcal{E}_2 \in O[\![(\textbf{let } \Pi \textbf{ in } \mathsf{MGCp}_n), (\sigma_{\mathsf{MGC}(n)}, \sigma_o, \circledcirc)]\!]$ and $\mathcal{E}_1 \approx \mathcal{E}_2$.
(b) For any $\mathcal{E}_2$, if $\mathcal{E}_2 \in O[\![(\textbf{let } \Pi \textbf{ in } \mathsf{MGCp}_n), (\sigma_{\mathsf{MGC}(n)}, \sigma_a, \circledcirc)]\!]$, then there exists $\mathcal{E}_1$ such that $\mathcal{E}_1 \in \mathcal{H}[\![(\textbf{let } \Pi \textbf{ in } \mathsf{MGC}_n), (\sigma_{\mathsf{MGC}(n)}, \sigma_a, \circledcirc)]\!]$ and $\mathcal{E}_1 \approx \mathcal{E}_2$.
Here $\mathcal{E}_1 \approx \mathcal{E}_2$ is inductively defined as follows.

$$\frac{}{\epsilon \approx \epsilon} \qquad \frac{e_1 \approx e_2 \qquad \mathcal{E}_1 \approx \mathcal{E}_2}{e_1 :: \mathcal{E}_1 \approx e_2 :: \mathcal{E}_2}$$

$$\frac{}{(\mathsf{t}, f_i, n) \approx (\mathsf{t}, \textbf{out}, (i, n))} \qquad \frac{}{(\mathsf{t}, \textbf{ret}, n) \approx (\mathsf{t}, \textbf{out}, n)}$$

$$\frac{}{(\mathsf{t}, \textbf{obj}, \textbf{abort}) \approx (\mathsf{t}, \textbf{obj}, \textbf{abort})}$$

Proof. By constructing simulations between MGC_n and MGCp_n. □

(2) $\Pi \subsetneq_P \Gamma \implies \Pi \sqsubseteq_P^{\mathsf{fin}} \Gamma$:

Proof. For any $n, C_1, \ldots, C_n, \sigma_c, \sigma_{\mathsf{MGC}(n)}$ and σ_o, by Lemma B.46, we know

$$\mathcal{H}[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_o, \circledcirc)]\!] \subseteq \mathcal{H}[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ \mathsf{MGC}_n), (\sigma_{\mathsf{MGC}(n)}, \sigma_o, \circledcirc)]\!].$$

Since $\Pi \subsetneq_P \Gamma$, we know for any σ_a such that $(\sigma_o, \sigma_a) \models P$, we have

$$\mathcal{H}[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ \mathsf{MGC}_n), (\sigma_{\mathsf{MGC}(n)}, \sigma_o, \circledcirc)]\!] \subseteq \mathcal{H}[\![(\mathbf{let}\ \Gamma\ \mathbf{in}\ \mathsf{MGC}_n), (\sigma_{\mathsf{MGC}(n)}, \sigma_a, \circledcirc)]\!].$$

Thus, for any $\mathcal{E}$ such that

$$\mathcal{E} \in \mathcal{T}[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_o, \circledcirc)]\!],$$

there exists $\mathcal{E}_{\mathsf{MGC}}$ such that $\mathsf{get\_hist}(\mathcal{E}) = \mathsf{get\_hist}(\mathcal{E}_{\mathsf{MGC}})$ and

$$\mathcal{E}_{\mathsf{MGC}} \in \mathcal{T}[\![(\mathbf{let}\ \Gamma\ \mathbf{in}\ \mathsf{MGC}_n), (\sigma_{\mathsf{MGC}(n)}, \sigma_a, \circledcirc)]\!].$$

Intuitively, we can then replace the object events in $\mathcal{E}$ with those in $\mathcal{E}_{\mathsf{MGC}}$ and keep other events (and the order between them) unchanged. Thus the resulting trace $\mathcal{E}'$ satisfies $\mathsf{get\_obsv}(\mathcal{E}') = \mathsf{get\_obsv}(\mathcal{E})$. We prove $\mathcal{E}'$ can be produced by the corresponding abstract client program, that is

$$\mathcal{E}' \in \mathcal{T}[\![(\mathbf{let}\ \Gamma\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_a, \circledcirc)]\!].$$

Then we are done.

Alternatively, we can actually construct a "simulation" relation $\lesssim$ (see Fig. 30(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.11).

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

(1) if $(W_1, \mathcal{S}_1) \overset{e_1}{\longmapsto} \mathbf{abort}$, then there exists $\mathcal{E}_3$ such that $(W_3, \mathcal{S}_3) \overset{\mathcal{E}_3}{\longmapsto}{}^* \mathbf{abort}$ and $e_1 = \mathsf{get\_obsv}(\mathcal{E}_3)$;

(2) if $(W_1, \mathcal{S}_1) \overset{e_1}{\longmapsto} (W_1', \mathcal{S}_1')$ and $\mathsf{is\_clt}(e_1)$, then there exist W_3' and $\mathcal{S}_3'$ such that $(W_3, \mathcal{S}_3) \overset{e_1}{\longmapsto} (W_3', \mathcal{S}_3')$ and $(W_1', \mathcal{S}_1') \lesssim (W_2, \mathcal{S}_2; W_3', \mathcal{S}_3')$.

(3) if $(W_1, \mathcal{S}_1) \overset{e_1}{\longmapsto} (W_1', \mathcal{S}_1')$ and $\mathsf{is\_obj}(e_1)$, then there exist $\mathcal{E}_2, W_2', \mathcal{S}_2', \mathcal{E}_3, W_3'$ and $\mathcal{S}_3'$ such that $(W_2, \mathcal{S}_2) \overset{\mathcal{E}_2}{\longmapsto}{}^* (W_2', \mathcal{S}_2')$, $(W_3, \mathcal{S}_3) \overset{\mathcal{E}_3}{\longmapsto}{}^* (W_3', \mathcal{S}_3')$, $\mathsf{get\_hist}(e_1) = \mathsf{get\_hist}(\mathcal{E}_2)$, $\mathsf{get\_obj}(\mathcal{E}_2) = \mathcal{E}_3$ and $(W_1', \mathcal{S}_1') \lesssim (W_2', \mathcal{S}_2'; W_3', \mathcal{S}_3')$.

(B.11)

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 $\mathsf{get\_obj}(\mathcal{E})$ to get the sub-trace of $\mathcal{E}$ consisting of object events only. We can prove the following from (B.11).

$$\text{If } (W_1, \mathcal{S}_1) \lesssim (W_2, \mathcal{S}_2; W_3, \mathcal{S}_3), \text{ then } O[\![W_1, \mathcal{S}_1]\!] \subseteq O[\![W_3, \mathcal{S}_3]\!].$$

Initially,

$$(\mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n, (\sigma_c, \sigma_o, \circledcirc)) \lesssim (\mathbf{let}\ \Gamma\ \mathbf{in}\ \mathsf{MGC}_n, (\sigma_{\mathsf{MGC}(n)}, \sigma_a, \circledcirc);\ \mathbf{let}\ \Gamma\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n, (\sigma_c, \sigma_a, \circledcirc))$$

holds, and we are done. □

$$
\begin{array}{l}
(\textbf{let}\ \Pi\ \textbf{in}\ C_1 \parallel \ldots \parallel C_n, (\sigma_c, \sigma_o, \{1 \rightsquigarrow \kappa_1, \ldots, n \rightsquigarrow \kappa_n\})) \\
\lesssim_{\mathsf{MGC}} (\textbf{let}\ \Pi\ \textbf{in}\ C'_1 \parallel \ldots \parallel C'_n, (\sigma_{\mathsf{MGC}(n)}, \sigma_o, \{1 \rightsquigarrow \kappa'_1, \ldots, n \rightsquigarrow \kappa'_n\})) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{where } \forall i.\ (C_i, \kappa_i) \lesssim^{i}_{\mathsf{MGC}} (C'_i, \kappa'_i)
\end{array}
$$

$$
(C, \circ) \lesssim^{\mathsf{t}}_{\mathsf{MGC}} (\mathsf{MGT}_{\mathsf{t}}, \circ) \qquad (C, (\sigma_l, x, C')) \lesssim^{\mathsf{t}}_{\mathsf{MGC}} (C, (\sigma_l, z_{\mathsf{t}}, (\textbf{skip}; \mathsf{MGT}_{\mathsf{t}})))
$$

(a) the program is simulated by MGC

$$
\begin{array}{l}
(\textbf{let}\ \Pi\ \textbf{in}\ C_1 \parallel \ldots \parallel C_n, (\sigma_c, \sigma_o, \{1 \rightsquigarrow \kappa_1, \ldots, n \rightsquigarrow \kappa_n\})) \\
\lesssim (\textbf{let}\ \Gamma\ \textbf{in}\ C'_1 \parallel \ldots \parallel C'_n, (\sigma_{\mathsf{MGC}(n)}, \sigma'_o, \{1 \rightsquigarrow \kappa'_1, \ldots, n \rightsquigarrow \kappa'_n\}); \\
\quad \textbf{let}\ \Gamma\ \textbf{in}\ C''_1 \parallel \ldots \parallel C''_n, (\sigma_c, \sigma'_o, \{1 \rightsquigarrow \kappa''_1, \ldots, n \rightsquigarrow \kappa''_n\})) \\
\text{where } \forall i.\ (C_i, \kappa_i) \lesssim (C'_i, \kappa'_i; C''_i, \kappa''_i) \\
\text{and } \mathcal{H}[\![\textbf{let}\ \Pi\ \textbf{in}\ C_1 \parallel \ldots \parallel C_n, (\sigma_c, \sigma_o, \{1 \rightsquigarrow \kappa_1, \ldots, n \rightsquigarrow \kappa_n\})]\!] \\
\quad \subseteq \mathcal{H}[\![\textbf{let}\ \Gamma\ \textbf{in}\ C'_1 \parallel \ldots \parallel C'_n, (\sigma_{\mathsf{MGC}(n)}, \sigma'_o, \{1 \rightsquigarrow \kappa'_1, \ldots, n \rightsquigarrow \kappa'_n\})]\!]
\end{array}
$$

$$
(C, \circ) \lesssim (C', \circ; C, \circ) \qquad (C, (\sigma_l, x, C_c)) \lesssim (C', (\sigma'_l, x', C'_c); C', (\sigma'_l, x, C_c))
$$

(b) the concrete program is "simulated" by the abstract MGC and the abstract program

Fig. 30. Simulations between programs and MGC.

Proof of Lemma B.45.

(1) $\Pi \subseteq_P \Gamma \implies \Pi \preceq_P^{\mathsf{MGC}} \Gamma$:
The premise $\Pi \subseteq_P \Gamma$ 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 B.48 (Γ is linearizable).
For any n, $\sigma_{\mathsf{MGC}(n)}$, σ_a and $\mathcal{E}$, if $\mathcal{E} \in \mathcal{H}[\![(\textbf{let}\ \Gamma\ \textbf{in}\ \mathsf{MGC}_n), (\sigma_{\mathsf{MGC}(n)}, \sigma_a, \circledcirc)]\!]$, then there exist $\mathcal{E}_c$ and $\mathcal{E}'$ such that $\mathcal{E}_c \in \mathsf{completions}(\mathcal{E})$, $\mathcal{E}_c \preceq^{\mathsf{lin}} \mathcal{E}'$, $\mathsf{seq}(\mathcal{E}')$ and $\mathcal{E}' \in \mathcal{H}[\![(\textbf{let}\ \Gamma\ \textbf{in}\ \mathsf{MGC}_n), (\sigma_{\mathsf{MGC}(n)}, \sigma_a, \circledcirc)]\!]$.

Proof. From the execution that generates $\mathcal{E}$, 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 $\mathcal{E}'$ satisfying all the requirements. □

(2) $\Pi \preceq_P^{\mathsf{MGC}} \Gamma \implies \Pi \subseteq_P \Gamma$:
The key is to prove that every linearizable history can be generated by the MGC with the *abstract* object Γ.

Lemma B.49 (Rearrangement).
For any n, $\sigma_{\mathsf{MGC}(n)}$, σ_a, $\mathcal{E}$ and $\mathcal{E}'$, if $\mathcal{E} \preceq^{\mathsf{lin}} \mathcal{E}'$, $\mathsf{seq}(\mathcal{E}')$ and $\mathcal{E}' \in \mathcal{H}[\![(\textbf{let}\ \Gamma\ \textbf{in}\ \mathsf{MGC}_n), (\sigma_{\mathsf{MGC}(n)}, \sigma_a, \circledcirc)]\!]$, then $\mathcal{E} \in \mathcal{H}[\![(\textbf{let}\ \Gamma\ \textbf{in}\ \mathsf{MGC}_n), (\sigma_{\mathsf{MGC}(n)}, \sigma_a, \circledcirc)]\!]$.

Proof. We construct the execution generating $\mathcal{E}$, 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 $\mathcal{E}'$. The detailed proof is by induction over the length of $\mathcal{E}$. □

B.6.3 Equivalence for starvation-freedom. By Theorem B.38, 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 B.50. $\mathsf{SF\text{-}MGC}_P(\Pi)$, iff

$$\forall n, \sigma_o, \mathcal{E}.\ \mathcal{E} \in \mathcal{T}_\omega[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ \mathsf{MGCp1}_n), (\sigma_{\mathsf{MGC}(n)}, \sigma_o, \circledcirc)]\!] \wedge \mathsf{fair}(\mathcal{E}) \wedge ((\sigma_o, \_) \models P) \implies \mathsf{prog\text{-}t}(\mathcal{E}) \vee \mathsf{abt}(\mathcal{E})\,.$$

We only need to prove the following lemmas.

Lemma B.51. $\Pi \sqsubseteq_P \Gamma \implies \Pi \sqsubseteq_P^{\mathsf{fin}} \Gamma\,.$

Lemma B.52. $\Pi \sqsubseteq_P \Gamma \implies \mathsf{SF\text{-}MGC}_P(\Pi)\,.$

Lemma B.53. $\mathsf{SF\text{-}MGC}_P(\Pi) \implies \mathsf{starvation\text{-}free}_P(\Pi)\,.$

Lemma B.54. $\Pi \sqsubseteq_P^{\mathsf{fin}} \Gamma \wedge \mathsf{starvation\text{-}free}_P(\Pi) \implies \Pi \sqsubseteq_P \Gamma\,.$

Proof of Lemma B.51. For any $n, C_1, \ldots, C_n, \sigma_c, \sigma_o$ and σ_a such that $(\sigma_o, \sigma_a) \models P$, for any $\mathcal{E}$, if

$$\mathcal{E} \in O[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_o, \circledcirc)]\!],$$

we know there exists $\mathcal{E}_1$ such that $\mathcal{E} = \mathsf{get\_obsv}(\mathcal{E}_1)$ and

$$\mathcal{E}_1 \in \mathcal{T}[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_o, \circledcirc)]\!]\,.$$

We can prove: there exists $\mathcal{E}_1'$ and $\mathcal{E}_1''$ such that

$$\mathcal{E}_1'' = (\mathbf{spawn}, n) :: \mathcal{E}_1 :: \mathcal{E}_1'\ ,\quad \mathsf{fair}(\mathcal{E}_1'')\quad \text{and}\quad \mathcal{E}_1'' \in \mathcal{T}_\omega[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_o, \circledcirc)]\!].$$

Since $\Pi \sqsubseteq_P \Gamma$, we know there exists $\mathcal{E}_2''$ such that

$$\mathcal{E}_2'' \in \mathcal{T}_\omega[\![(\mathbf{let}\ \Gamma\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_a, \circledcirc)]\!]\ ,\quad \mathsf{fair}(\mathcal{E}_2'')\quad \text{and}\quad \mathsf{get\_obsv}(\mathcal{E}_2'') = \mathsf{get\_obsv}(\mathcal{E}_1'') = \mathcal{E} :: \mathsf{get\_obsv}(\mathcal{E}_1')\,.$$

Thus there exists $\mathcal{E}_2$ such that

$$\mathcal{E}_2 \in \mathcal{T}[\![(\mathbf{let}\ \Gamma\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_a, \circledcirc)]\!]\quad \text{and}\quad \mathsf{get\_obsv}(\mathcal{E}_2) = \mathcal{E}.$$

Thus $\mathcal{E} \in O[\![(\mathbf{let}\ \Gamma\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_a, \circledcirc)]\!]$ and we are done.

Proof of Lemma B.52. For any n, σ_o, σ_a and $\mathcal{E}$ such that $\mathcal{E} \in \mathcal{T}_\omega[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ \mathsf{MGCp1}_n), (\sigma_{\mathsf{MGC}(n)}, \sigma_o, \circledcirc)]\!]$, $\mathsf{fair}(\mathcal{E})$ and $(\sigma_o, \sigma_a) \models P$, suppose

$$\neg\mathsf{abt}(\mathcal{E})\,,$$

then by the operational semantics, we only need to prove:

for any i and e, if $e \in \mathsf{pend\_inv}(\mathcal{E}(1..i))$, then there exists $j > i$ such that $\mathsf{match}(e, \mathcal{E}(j))$

.

Suppose it does not hold. Then we know there exists t_0 such that

$$\exists i.\ \forall j.\ j \geq i \Rightarrow (\mathcal{E}|_{\mathsf{t}_0})(j) = (\mathsf{t}_0, \mathbf{obj})\,.$$

Thus we have

$$|(\mathsf{get\_obsv}(\mathcal{E})|_{\mathsf{t}_0})| < i\quad \text{and}\quad \mathsf{last}(\mathsf{get\_obsv}(\mathcal{E})|_{\mathsf{t}_0}) = (\mathsf{t}_0, \mathbf{out}, 0)\ .$$

Since $\Pi \sqsubseteq_P \Gamma$, we know:

$$O_{\mathit{fo}}[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ \mathsf{MGCp1}_n), (\sigma_{\mathsf{MGC}(n)}, \sigma_o, \circledcirc)]\!] \subseteq O_{\mathit{fo}}[\![(\mathbf{let}\ \Gamma\ \mathbf{in}\ \mathsf{MGCp1}_n), (\sigma_{\mathsf{MGC}(n)}, \sigma_a, \circledcirc)]\!]\,.$$

Thus there exists $\mathcal{E}'$ such that

$$\mathcal{E}' \in \mathcal{T}_\omega[\![(\mathbf{let}\ \Gamma\ \mathbf{in}\ \mathsf{MGCp1}_n), (\sigma_{\mathsf{MGC}(n)}, \sigma_a, \circledcirc)]\!]\,,\quad \mathsf{fair}(\mathcal{E}')\quad \text{and}\quad \mathsf{get\_obsv}(\mathcal{E}') = \mathsf{get\_obsv}(\mathcal{E})\,.$$

Thus $\mathsf{last}(\mathsf{get\_obsv}(\mathcal{E}')|_{\mathsf{t}_0}) = (\mathsf{t}_0, \mathbf{out}, 0)$. But this is impossible from the operational semantics, $\mathsf{fair}(\mathcal{E}')$ and the fact that Γ is atomic and total. Thus we are done.

Proof of Lemma B.53. Similar to the proof of Lemma B.43, we first prove the following two lemmas (Lemma B.55 and Lemma B.56). Then, from SF-MGC$_P(\Pi)$, we know:
if $\mathcal{E}' \in \mathcal{T}_\omega[\![(\textbf{let } \Pi \textbf{ in } \mathsf{MGCp1}_n), (\sigma_{\mathsf{MGC}(n)}, \sigma_o, \circledcirc)]\!]$, $(\sigma_o, \_) \models P$ and fair$(\mathcal{E}')$, then either prog-t$(\mathcal{E}')$ or abt$(\mathcal{E}')$. By the definition of $\mathsf{MGCp1}_n$ and the operational semantics, we know either prog-t$(\mathcal{E}')$ or obj_abt$(\mathcal{E}')$. From Lemmas B.55 and B.56, we know: if $\mathcal{E} \in \mathcal{T}_\omega[\![(\textbf{let } \Pi \textbf{ in } C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_o, \circledcirc)]\!]$, $(\sigma_o, \_) \models P$ and fair$(\mathcal{E})$, then either clt_abt$(\mathcal{E})$, or prog-t$(\mathcal{E})$, or obj_abt$(\mathcal{E})$. Thus we have starvation-free$_P(\Pi)$.

Lemma B.55 (MGC is the most general). *For any n, $C_1, \ldots, C_n$, σ_c, σ_o and $\mathcal{E}$, if*

$$\mathcal{E} \in \mathcal{T}_\omega[\![(\textbf{let } \Pi \textbf{ in } C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_o, \circledcirc)]\!] \text{ and } \mathsf{fair}(\mathcal{E}),$$

then one of the following holds:

(1) clt_abt$(\mathcal{E})$*; or*
(2) there exists $\mathcal{E}'$ such that

$$(\mathcal{E}' \in \mathcal{T}_\omega[\![(\textbf{let } \Pi \textbf{ in } \mathsf{MGCp1}_n), (\sigma_{\mathsf{MGC}(n)}, \sigma_o, \circledcirc)]\!]) \wedge (\mathsf{get\_obj}(\mathcal{E}) = \mathsf{get\_obj}(\mathcal{E}')) \wedge \mathsf{fair}(\mathcal{E}').$$

Here get_obj$(\mathcal{E})$ *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 $\mathcal{E}$ for the client at the left side. □

Lemma B.56. *For any $\mathcal{E}$ and $\mathcal{E}'$, if* get_obj$(\mathcal{E})$ = get_obj$(\mathcal{E}')$ *and* prog-t$(\mathcal{E}')$*, then* prog-t$(\mathcal{E})$.

Proof. By unfolding the definitions. □

Proof of Lemma B.54. The key is to show the following (B.12).

For any $n, C_1, \ldots, C_n, \sigma_c, \sigma_o$ and σ_a such that $(\sigma_o, \sigma_a) \models P$,
if $(\lfloor \textbf{let } \Pi \textbf{ in } C_1 \,\|\, \ldots \,\|\, C_n \rfloor, (\sigma_c, \sigma_o, \circledcirc)) \overset{\mathcal{E}}{\longmapsto}^\omega \cdot$ and fair$(\mathcal{E})$, then there exists $\mathcal{E}_a$ such that
$(\lfloor \textbf{let } \Gamma \textbf{ in } C_1 \,\|\, \ldots \,\|\, C_n \rfloor, (\sigma_c, \sigma_a, \circledcirc)) \overset{\mathcal{E}_a}{\longmapsto}^\omega \cdot$, get_obsv$(\mathcal{E})$ = get_obsv$(\mathcal{E}_a)$ and fair$(\mathcal{E}_a)$. (B.12)

Its proof is similar to the proof of $\Pi \subseteq_P \Gamma \Longrightarrow \Pi \sqsubseteq_P^{\mathsf{fin}} \Gamma$. We can replace the object events in the concrete *infinite* trace $\mathcal{E}$ with those generated by MGC using Γ. The resulting trace $\mathcal{E}_a$ satisfies get_obsv$(\mathcal{E}_a)$ = get_obsv$(\mathcal{E})$. From starvation-free$_P(\Pi)$, we can show $\mathcal{E}_a$ must be infinite and fair.

In detail, we construct a "simulation" relation $\lesssim$ (see Fig. 30(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.13). (It is derived from (B.11).) Here we use $\mathcal{E}\backslash(\_, \textbf{obj})$ for a sub-trace resulting from removing all the events of the form $(\_, \textbf{obj})$ in $\mathcal{E}$.

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 $\mathcal{E}_2, W_2', \mathcal{S}_2', \mathcal{E}_3, W_3'$ and $\mathcal{S}_3'$ such that
$(W_2, \mathcal{S}_2) \overset{\mathcal{E}_2}{\longmapsto}^* (W_2', \mathcal{S}_2')$, $(W_3, \mathcal{S}_3) \overset{\mathcal{E}_3}{\longmapsto}^* (W_3', \mathcal{S}_3')$,
$\mathcal{E}_3\backslash(\_, \textbf{obj}) = e_1\backslash(\_, \textbf{obj})$ and $(W_1', \mathcal{S}_1') \lesssim (W_2', \mathcal{S}_2'; W_3', \mathcal{S}_3')$. (B.13)

With (B.13), we can prove the following (B.14) by induction over the length of $\mathcal{E}_1$.

$$\begin{array}{l}\text{For any } W_1, \mathcal{S}_1, W_2, \mathcal{S}_2, W_3, \mathcal{S}_3 \text{ and } \mathcal{E}_1,\\ \text{if } (W_1, \mathcal{S}_1) \lesssim (W_2, \mathcal{S}_2; W_3, \mathcal{S}_3),\ (W_1, \mathcal{S}_1) \xmapsto{\mathcal{E}_1}{}^{+} (W_1', \mathcal{S}_1') \text{ and } \mathsf{last}(\mathcal{E}_1) \neq (\_, \mathbf{obj}),\\ \text{then there exist } \mathcal{E}_2, W_2', \mathcal{S}_2', \mathcal{E}_3, W_3' \text{ and } \mathcal{S}_3' \text{ such that}\\ (W_2, \mathcal{S}_2) \xmapsto{\mathcal{E}_2}{}^{*} (W_2', \mathcal{S}_2'),\ (W_3, \mathcal{S}_3) \xmapsto{\mathcal{E}_3}{}^{+} (W_3', \mathcal{S}_3'),\\ \mathcal{E}_1 \backslash (\_, \mathbf{obj}) = \mathcal{E}_3 \backslash (\_, \mathbf{obj}) \text{ and } (W_1', \mathcal{S}_1') \lesssim (W_2', \mathcal{S}_2'; W_3', \mathcal{S}_3').\end{array} \tag{B.14}$$

Then we prove the following (B.15) by co-induction.

$$\begin{array}{l}\text{For any } n, C_1, \ldots, C_n, \sigma_c, \sigma_o, W_1, \mathcal{S}_1, W_2, \mathcal{S}_2, W_3, \mathcal{S}_3, \mathcal{E}_0 \text{ and } \mathcal{E}_1,\\ \text{if } (\lfloor \mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \parallel \ldots \parallel C_n \rfloor, (\sigma_c, \sigma_o, \circledcirc)) \xmapsto{\mathcal{E}_0}{}^{*} (W_1, \mathcal{S}_1),\\ (W_1, \mathcal{S}_1) \lesssim (W_2, \mathcal{S}_2; W_3, \mathcal{S}_3),\ (W_1, \mathcal{S}_1) \xmapsto{\mathcal{E}_1}{}^{\omega} \cdot \text{ and } \mathsf{prog\text{-}t}(\mathcal{E}_0 :: \mathcal{E}_1),\\ \text{then there exists } \mathcal{E}_3 \text{ such that } (W_3, \mathcal{S}_3) \xmapsto{\mathcal{E}_3}{}^{\omega} \cdot \text{ and } \mathcal{E}_1 \backslash (\_, \mathbf{obj}) = \mathcal{E}_3 \backslash (\_, \mathbf{obj}).\end{array} \tag{B.15}$$

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

$$\begin{array}{l}(\lfloor \mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \parallel \ldots \parallel C_n \rfloor, (\sigma_c, \sigma_o, \circledcirc))\\ \lesssim (\mathbf{let}\ \Gamma\ \mathbf{in}\ \mathsf{MGC}_n, (\sigma_{\mathsf{MGC}(n)}, \sigma_a, \circledcirc);\ \lfloor \mathbf{let}\ \Gamma\ \mathbf{in}\ C_1 \parallel \ldots \parallel C_n \rfloor, (\sigma_c, \sigma_a, \circledcirc))\ .\end{array}$$

If $(\lfloor \mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \parallel \ldots \parallel C_n \rfloor, (\sigma_c, \sigma_o, \circledcirc)) \xmapsto{\mathcal{E}}{}^{\omega} \cdot$ and $\mathsf{fair}(\mathcal{E})$, by starvation-free$_P(\Pi)$, we know $\mathsf{prog\text{-}t}(\mathcal{E})$. Then from (B.15) we get: there exists $\mathcal{E}_a$ such that

$$(\lfloor \mathbf{let}\ \Gamma\ \mathbf{in}\ C_1 \parallel \ldots \parallel C_n \rfloor, (\sigma_c, \sigma_a, \circledcirc)) \xmapsto{\mathcal{E}_a}{}^{\omega} \cdot, \text{ and } \mathcal{E} \backslash (\_, \mathbf{obj}) = \mathcal{E}_a \backslash (\_, \mathbf{obj}).$$

Thus we know $\mathsf{get\_obsv}(\mathcal{E}) = \mathsf{get\_obsv}(\mathcal{E}_a)$.

Below we prove $\mathsf{fair}(\mathcal{E}_a)$. Since $\mathsf{fair}(\mathcal{E})$ and $|\mathcal{E}| = \omega$, we know for any t,

$$\text{either } |(\mathcal{E}|_{\mathrm{t}})| = \omega, \quad \text{or } \mathsf{last}(\mathcal{E}|_{\mathrm{t}}) = (\mathrm{t}, \mathbf{term})\ .$$

(a) $\mathsf{last}(\mathcal{E}|_{\mathrm{t}}) = (\mathrm{t}, \mathbf{term})$:
Since $\mathcal{E} \backslash (\_, \mathbf{obj}) = \mathcal{E}_a \backslash (\_, \mathbf{obj})$ and by the operational semantics, we know $\mathsf{last}(\mathcal{E}_a|_{\mathrm{t}}) = (\mathrm{t}, \mathbf{term})$.

(b) $|(\mathcal{E}|_{\mathrm{t}})| = \omega$:
Since $\mathcal{E} \backslash (\_, \mathbf{obj}) = \mathcal{E}_a \backslash (\_, \mathbf{obj})$, we know

$$(\mathcal{E}|_{\mathrm{t}}) \backslash (\mathrm{t}, \mathbf{obj}) = (\mathcal{E}_a|_{\mathrm{t}}) \backslash (\mathrm{t}, \mathbf{obj})\ .$$

Suppose $|(\mathcal{E}_a|_{\mathrm{t}})| \neq \omega$. Then we know $|\ ((\mathcal{E}_a|_{\mathrm{t}}) \backslash (\mathrm{t}, \mathbf{obj}))\ | \neq \omega$. Thus

$$\exists i.\ \forall j.\ j \geq i\ \Rightarrow\ (\mathcal{E}|_{\mathrm{t}})(j) = (\mathrm{t}, \mathbf{obj})\ .$$

By the operational semantics, we know there exists i such that

$$\mathsf{tid}(\mathcal{E}(i)) = \mathrm{t}\ ,\quad \mathsf{is\_inv}(\mathcal{E}(i))\ ,\quad \text{and}\quad \forall j.\ j \geq i\ \Rightarrow\ \neg\mathsf{match}(\mathcal{E}(i), \mathcal{E}(j))\ .$$

This contradicts the premise $\mathsf{prog\text{-}t}(\mathcal{E})$. Thus we know $|(\mathcal{E}_a|_{\mathrm{t}})| = \omega$.

Thus $\mathsf{fair}(\mathcal{E}_a)$ holds and we are done.

B.6.4 Equivalence for deadlock-freedom. We first define the following contextual refinement $\widehat{\sqsubseteq}$ which assumes fair scheduling at the concrete side only.

Definition B.57. $\Pi\ \widehat{\sqsubseteq}_P\ \Gamma$ iff

$$\begin{array}{l}\forall n, C_1, \ldots, C_n, \sigma_c, \sigma, \Sigma.\ ((\sigma, \Sigma) \models P) \Longrightarrow\\ O_{\mathit{fo}}[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \parallel \ldots \parallel C_n), (\sigma_c, \sigma, \circledcirc)]\!] \subseteq O_{(\mathbf{let}\ \Gamma\ \mathbf{in}\ C_1 \parallel \ldots \parallel C_n), (\sigma_c, \Sigma, \circledcirc)}[\![\ ]\!].\end{array}$$

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 $\mathsf{wr}_1(id)$ adds the shared variable 1 to the higher-level states, and $\mathsf{wr}_1^{-1}(id)$ removes 1 which should be 0:

$$(\sigma, \Sigma) \models \mathsf{wr}_1(id) \quad \text{iff} \quad \Sigma = \sigma \uplus \{1 \rightsquigarrow 0\}$$
$$(\sigma, \Sigma) \models \mathsf{wr}_1^{-1}(id) \quad \text{iff} \quad \sigma = \Sigma \uplus \{1 \rightsquigarrow 0\}$$

To prove Theorem B.40, we prove the following:

$$\Pi \sqsubseteq_P^{\mathsf{fin}} \Gamma \wedge \mathsf{deadlock\text{-}free}_P(\Pi) \iff \Pi \mathrel{\widehat{\sqsubseteq}}_P \Gamma \tag{B.16}$$

$$\Pi \sqsubseteq_{\mathsf{wr}_1(P)} \mathsf{wr}_1(\Gamma) \implies \Pi \mathrel{\widehat{\sqsubseteq}}_P \Gamma \tag{B.17}$$

$$\Pi \sqsubseteq_P^{\mathsf{fin}} \Gamma \wedge \mathsf{deadlock\text{-}free}_P(\Pi) \implies \Pi \sqsubseteq_{\mathsf{wr}_1(P)} \mathsf{wr}_1(\Gamma) \tag{B.18}$$

Proofs of (B.16). We first define the MGC version of deadlock-freedom.

Definition B.58. $\mathsf{deadlock\text{-}free}_P^{\mathsf{MGC}}(\Pi)$, iff

$\forall n, \sigma_o, \mathcal{E}.\ \mathcal{E} \in \mathcal{T}_\omega[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ \mathsf{MGCp1}_n), (\sigma_{\mathsf{MGC}(n)}, \sigma_o, \circledcirc)]\!] \wedge \mathsf{fair}(\mathcal{E}) \wedge ((\sigma_o, \_) \models P)$
$\implies \mathsf{prog\text{-}p}(\mathcal{E}) \vee \mathsf{abt}(\mathcal{E})$.

We only need to prove the following lemmas.

Lemma B.59. $\Pi \mathrel{\widehat{\sqsubseteq}}_P \Gamma \implies \Pi \sqsubseteq_P^{\mathsf{fin}} \Gamma$.

Lemma B.60. $\Pi \mathrel{\widehat{\sqsubseteq}}_P \Gamma \implies \mathsf{deadlock\text{-}free}_P^{\mathsf{MGC}}(\Pi)$.

Lemma B.61. $\mathsf{deadlock\text{-}free}_P^{\mathsf{MGC}}(\Pi) \implies \mathsf{deadlock\text{-}free}_P(\Pi)$.

Lemma B.62. $\Pi \sqsubseteq_P^{\mathsf{fin}} \Gamma \wedge \mathsf{deadlock\text{-}free}_P(\Pi) \implies \Pi \mathrel{\widehat{\sqsubseteq}}_P \Gamma$.

Proof of Lemma B.59. Similar to the proof of Lemma B.51. □

Proof of Lemma B.60. Similar to the proof of Lemma B.52. For any n, σ_o, σ_a and $\mathcal{E}$ such that $\mathcal{E} \in \mathcal{T}_\omega[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ \mathsf{MGCp1}_n), (\sigma_{\mathsf{MGC}(n)}, \sigma_o, \circledcirc)]\!]$, $\mathsf{fair}(\mathcal{E})$ and $(\sigma_o, \sigma_a) \models P$, suppose

$$\neg \mathsf{abt}(\mathcal{E}) ,$$

then by the operational semantics, we only need to prove:

> for any i and e, if $e \in \mathsf{pend\_inv}(\mathcal{E}(1..i))$, then there exists $j > i$ such that $\mathsf{is\_ret}(\mathcal{E}(j))$ holds.

Suppose it does not hold. Then we know:

$$\exists i.\ \forall j.\ j \geq i \Rightarrow \mathcal{E}(j) = (\_, \mathbf{obj}) .$$

Since $\mathsf{fair}(\mathcal{E})$, we have

$$\forall \mathsf{t}_0 \in [1..n].\ \mathsf{last}(\mathsf{get\_obsv}(\mathcal{E})|_{\mathsf{t}_0}) = (\mathsf{t}_0, \mathbf{out}, 0) \vee \mathsf{last}(\mathsf{get\_obsv}(\mathcal{E})|_{\mathsf{t}_0}) = (\mathsf{t}_0, \mathbf{out}, 2) \ .$$

Since $\Pi \mathrel{\widehat{\sqsubseteq}}_P \Gamma$, we know:

$$O_{fo}[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ \mathsf{MGCp1}_n), (\sigma_{\mathsf{MGC}(n)}, \sigma_o, \circledcirc)]\!] \subseteq O_{(\mathbf{let}\ \Gamma\ \mathbf{in}\ \mathsf{MGCp1}_n), (\sigma_{\mathsf{MGC}(n)}, \sigma_a, \circledcirc)}[\![\ ]\!].$$

Thus there exists $\mathcal{E}'$ such that

$$\mathcal{E}' \in \mathcal{T}_\omega[\![(\mathbf{let}\ \Gamma\ \mathbf{in}\ \mathsf{MGCp1}_n), (\sigma_{\mathsf{MGC}(n)}, \sigma_a, \circledcirc)]\!] \quad \text{and} \quad \mathsf{get\_obsv}(\mathcal{E}') = \mathsf{get\_obsv}(\mathcal{E}) .$$

Thus $\forall \mathsf{t}_0 \in [1..n].\ \mathsf{last}(\mathsf{get\_obsv}(\mathcal{E}')|_{\mathsf{t}_0}) = (\mathsf{t}_0, \mathbf{out}, 0) \vee \mathsf{last}(\mathsf{get\_obsv}(\mathcal{E}')|_{\mathsf{t}_0}) = (\mathsf{t}_0, \mathbf{out}, 2)$. But this is impossible from the operational semantics and the fact that Γ is atomic and total. Thus we are done. □

Proof of Lemma B.61. Similar to the proof of Lemma B.53. From $\mathsf{deadlock\text{-}free}_P^{\mathsf{MGC}}(\Pi)$, we get: if $\mathcal{E}' \in \mathcal{T}_\omega[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ \mathsf{MGCp1}_n), (\sigma_{\mathsf{MGC}(n)}, \sigma_o, \circledcirc)]\!], (\sigma_o, \_) \models P$ and $\mathsf{fair}(\mathcal{E}')$, then either $\mathsf{prog\text{-}p}(\mathcal{E}')$ or $\mathsf{abt}(\mathcal{E}')$. By the definition of $\mathsf{MGCp1}_n$ and the operational semantics, we know either $\mathsf{prog\text{-}p}(\mathcal{E}')$ or $\mathsf{obj\_abt}(\mathcal{E}')$. From Lemma B.55, we can prove: if $\mathcal{E} \in \mathcal{T}_\omega[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \parallel \ldots \parallel C_n), (\sigma_c, \sigma_o, \circledcirc)]\!]$, $(\sigma_o, \_) \models P$ and $\mathsf{fair}(\mathcal{E})$, then either $\mathsf{clt\_abt}(\mathcal{E})$, or $\mathsf{prog\text{-}p}(\mathcal{E})$, or $\mathsf{obj\_abt}(\mathcal{E})$. Thus we have $\mathsf{deadlock\text{-}free}_P(\Pi)$. □

Proofs of Lemma B.62. Similar to the proof of Lemma B.54. The key is to show the following (B.19).

$$\begin{array}{l}\text{For any } n, C_1, \ldots, C_n, \sigma_c, \sigma_o \text{ and } \sigma_a \text{ such that } (\sigma_o, \sigma_a) \models P,\\ \text{if } (\lfloor \mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \parallel \ldots \parallel C_n \rfloor, (\sigma_c, \sigma_o, \circledcirc)) \overset{\mathcal{E}}{\longmapsto}{}^{\omega} \cdot \text{ and } \mathsf{fair}(\mathcal{E}), \text{ then there exists } \mathcal{E}_a \text{ such that}\\ (\lfloor \mathbf{let}\ \Gamma\ \mathbf{in}\ C_1 \parallel \ldots \parallel C_n \rfloor, (\sigma_c, \sigma_a, \circledcirc)) \overset{\mathcal{E}_a}{\longmapsto}{}^{\omega} \cdot \text{ and } \mathsf{get\_obsv}(\mathcal{E}) = \mathsf{get\_obsv}(\mathcal{E}_a).\end{array} \tag{B.19}$$

As in the proofs for (B.12), we construct a "simulation" relation $\lesssim$ (see Fig. 30(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.20).

$$\begin{array}{l}\text{For any } n, C_1, \ldots, C_n, \sigma_c, \sigma_o, W_1, \mathcal{S}_1, W_2, \mathcal{S}_2, W_3, \mathcal{S}_3, \mathcal{E}_0 \text{ and } \mathcal{E}_1,\\ \text{if } (\lfloor \mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \parallel \ldots \parallel C_n \rfloor, (\sigma_c, \sigma_o, \circledcirc)) \overset{\mathcal{E}_0}{\longmapsto}{}^{*} (W_1, \mathcal{S}_1),\\ (W_1, \mathcal{S}_1) \lesssim (W_2, \mathcal{S}_2; W_3, \mathcal{S}_3), (W_1, \mathcal{S}_1) \overset{\mathcal{E}_1}{\longmapsto}{}^{\omega} \cdot \text{ and } \mathsf{prog\text{-}p}(\mathcal{E}_0 :: \mathcal{E}_1),\\ \text{then there exists } \mathcal{E}_3 \text{ such that } (W_3, \mathcal{S}_3) \overset{\mathcal{E}_3}{\longmapsto}{}^{\omega} \cdot \text{ and } \mathcal{E}_1 \backslash (\_, \mathbf{obj}) = \mathcal{E}_3 \backslash (\_, \mathbf{obj}).\end{array} \tag{B.20}$$

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

$$\begin{array}{l}(\lfloor \mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \parallel \ldots \parallel C_n \rfloor, (\sigma_c, \sigma_o, \circledcirc))\\ \lesssim (\mathbf{let}\ \Gamma\ \mathbf{in}\ \mathsf{MGC}_n, (\sigma_{\mathsf{MGC}(n)}, \sigma_a, \circledcirc);\ \lfloor \mathbf{let}\ \Gamma\ \mathbf{in}\ C_1 \parallel \ldots \parallel C_n \rfloor, (\sigma_c, \sigma_a, \circledcirc))\ .\end{array}$$

If $(\lfloor \mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \parallel \ldots \parallel C_n \rfloor, (\sigma_c, \sigma_o, \circledcirc)) \overset{\mathcal{E}}{\longmapsto}{}^{\omega} \cdot$ and $\mathsf{fair}(\mathcal{E})$, by $\mathsf{deadlock\text{-}free}_P(\Pi)$, we know $\mathsf{prog\text{-}p}(\mathcal{E})$. Then from (B.20) we get: there exists $\mathcal{E}_a$ such that

$$(\lfloor \mathbf{let}\ \Gamma\ \mathbf{in}\ C_1 \parallel \ldots \parallel C_n \rfloor, (\sigma_c, \sigma_a, \circledcirc)) \overset{\mathcal{E}_a}{\longmapsto}{}^{\omega} \cdot, \text{ and } \mathcal{E} \backslash (\_, \mathbf{obj}) = \mathcal{E}_a \backslash (\_, \mathbf{obj}).$$

Thus we know $\mathsf{get\_obsv}(\mathcal{E}) = \mathsf{get\_obsv}(\mathcal{E}_a)$ and we are done. □

Proofs of (B.17). We only need to prove the following:

$$\mathsf{wr}_1(\Gamma) \ \widehat{\sqsubseteq}_{\mathsf{wr}_1^{-1}(id)}\ \Gamma$$

By (B.16), we only need to show:

$$\mathsf{wr}_1(\Gamma) \ \sqsubseteq^{\mathsf{fin}}_{\mathsf{wr}_1^{-1}(id)}\ \Gamma \tag{B.21}$$

$$\mathsf{deadlock\text{-}free}_{\mathsf{wr}_1^{-1}(id)}(\mathsf{wr}_1(\Gamma)) \tag{B.22}$$

Proof of (B.21). We want to show: for any $n, C_1, \ldots, C_n, \sigma_c$ and σ_a,

$$O[\![(\textbf{let } \mathsf{wr}_1(\Gamma) \textbf{ in } C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_a \uplus \{1 \rightsquigarrow 0\}))]\!] \\ \subseteq O[\![(\textbf{let } \Gamma \textbf{ in } C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_a)]\!] \,.$$

We construct a simulation relation $\lesssim$ as follows.

$$(\textbf{let } \mathsf{wr}_1(\Gamma) \textbf{ in } C_1' \,\|\, \ldots \,\|\, C_n', (\sigma_c, \sigma_a', \{1 \rightsquigarrow \kappa_1', \ldots, n \rightsquigarrow \kappa_n'\})) \\ \lesssim (\textbf{let } \Gamma \textbf{ in } C_1 \,\|\, \ldots \,\|\, C_n, (\sigma_c, \sigma_a, \{1 \rightsquigarrow \kappa_1, \ldots, n \rightsquigarrow \kappa_n\})) \\ \text{if } \forall i.\ (C_i', \sigma_a', \kappa_i') \lesssim_i (C_i, \sigma_a, \kappa_i)$$

$$(C, \sigma_a \uplus \{1 \rightsquigarrow \_\}, \circ) \lesssim_{\mathsf{t}} (C, \sigma_a, \circ)$$

$$(\mathsf{wr}_1(\langle C \rangle); \mathsf{wr}_1(\textbf{return } E); \textbf{noret}, \sigma_a \uplus \{1 \rightsquigarrow \_\}, \kappa') \lesssim_{\mathsf{t}} (\langle C \rangle; \textbf{return } E; \textbf{noret}, \sigma_a, \kappa)$$

$$(\mathsf{wr}_1'(\langle C \rangle); \mathsf{wr}_1(\textbf{return } E); \textbf{noret}, \sigma_a \uplus \{1 \rightsquigarrow \_\}, \kappa') \lesssim_{\mathsf{t}} (\langle C \rangle; \textbf{return } E; \textbf{noret}, \sigma_a, \kappa)$$

$$(\mathsf{wr}_1''(\langle C \rangle); \mathsf{wr}_1(\textbf{return } E); \textbf{noret}, \sigma_a \uplus \{1 \rightsquigarrow \mathsf{t}\}, \kappa') \lesssim_{\mathsf{t}} (\langle C \rangle; \textbf{return } E; \textbf{noret}, \sigma_a, \kappa)$$

$$(\mathsf{wr}_1'''(\langle C \rangle); \mathsf{wr}_1(\textbf{return } E); \textbf{noret}, \sigma_a \uplus \{1 \rightsquigarrow \_\}, \kappa') \lesssim_{\mathsf{t}} (\langle C \rangle; \textbf{return } E; \textbf{noret}, \sigma_a, \kappa)$$

$$(\langle C \rangle; \mathsf{wr}_1(\textbf{return } E); \textbf{noret}, \sigma_a \uplus \{1 \rightsquigarrow \_\}, \kappa') \lesssim_{\mathsf{t}} (\langle C \rangle; \textbf{return } E; \textbf{noret}, \sigma_a, \kappa)$$

$$(\mathsf{wr}_1(\textbf{return } E); \textbf{noret}, \sigma_a \uplus \{1 \rightsquigarrow \_\}, \kappa') \lesssim_{\mathsf{t}} (\textbf{return } E; \textbf{noret}, \sigma_a, \kappa)$$

$$(\mathsf{wr}_1'(\textbf{return } E); \textbf{noret}, \sigma_a \uplus \{1 \rightsquigarrow \_\}, \kappa') \lesssim_{\mathsf{t}} (\textbf{return } E; \textbf{noret}, \sigma_a, \kappa)$$

$$(\mathsf{wr}_1''(\textbf{return } E); \textbf{noret}, \sigma_a \uplus \{1 \rightsquigarrow \mathsf{t}\}, \kappa') \lesssim_{\mathsf{t}} (\textbf{return } E; \textbf{noret}, \sigma_a, \kappa)$$

$$(\mathsf{wr}_1'''(\textbf{return } E); \textbf{noret}, \sigma_a \uplus \{1 \rightsquigarrow \_\}, \kappa') \lesssim_{\mathsf{t}} (\textbf{return } E; \textbf{noret}, \sigma_a, \kappa)$$

$$(\textbf{return } E; \textbf{noret}, \sigma_a \uplus \{1 \rightsquigarrow \_\}, \kappa') \lesssim_{\mathsf{t}} (\textbf{return } E; \textbf{noret}, \sigma_a, \kappa)$$

$$\text{where } \kappa' = (s_l \uplus \{\mathsf{u1} \rightsquigarrow \_, \mathsf{u2} \rightsquigarrow \_\}, x, C') \text{ , if } \kappa = (s_l, x, C')$$

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 $(W_B, \mathcal{S}_B) \overset{e_B}{\longmapsto} \textbf{abort}$, then
there exists $\mathcal{E}_A$ such that $(W_A, \mathcal{S}_A) \overset{\mathcal{E}_A}{\longmapsto}{}^* \textbf{abort}$ and $e_B = \mathsf{get\_obsv}(\mathcal{E}_A)$.

(2) If $(W_B, \mathcal{S}_B) \overset{e_B}{\longmapsto} (W_B', \mathcal{S}_B')$, then
there exist $\mathcal{E}_A$, W_A' and $\mathcal{S}_A'$ such that $(W_A, \mathcal{S}_A) \overset{\mathcal{E}_A}{\longmapsto}{}^* (W_A', \mathcal{S}_A')$, $\mathsf{get\_obsv}(e_B) = \mathsf{get\_obsv}(\mathcal{E}_A)$ and $(W_B', \mathcal{S}_B') \lesssim (W_A', \mathcal{S}_A')$.

(B.23)

With (B.23), we can prove the following by induction over the number of steps generating the event trace of $O[\![W_B, \mathcal{S}_B]\!]$.

If $(W_B, \mathcal{S}_B) \lesssim (W_A, \mathcal{S}_A)$ and $\mathcal{E} \in O[\![W_B, \mathcal{S}_B]\!]$, then $\mathcal{E} \in O[\![W_A, \mathcal{S}_A]\!]$. (B.24)

Since

$$(\textbf{let } \mathsf{wr}_1(\Gamma) \textbf{ in } C_1 \,\|\, \ldots \,\|\, C_n, (\sigma_c, \sigma_a \uplus \{1 \rightsquigarrow 0\}, \circledcirc)) \lesssim (\textbf{let } \Gamma \textbf{ in } C_1 \,\|\, \ldots \,\|\, C_n, (\sigma_c, \sigma_a, \circledcirc)) \,,$$

from (B.24), we are done. □

Proof of (B.22). We want to show: for any n, C_1, …, C_n, σ_c and σ_a,

$$\forall \mathcal{E}.\ \mathcal{E} \in \mathcal{T}_\omega[\![(\textbf{let } \mathsf{wr}_1(\Gamma) \textbf{ in } C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_a)]\!] \wedge (\sigma_a(1) = 0) \\ \Longrightarrow \mathsf{deadlock\text{-}free}(\mathcal{E}) \,.$$

It is reduced to the following.

$$\forall \mathcal{E}.\ \mathcal{E} \in \mathcal{T}_\omega[\![(\textbf{let } \mathsf{wr}_1(\Gamma) \textbf{ in } C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_a)]\!] \wedge (\sigma_a(1) = 0) \wedge \mathsf{fair}(\mathcal{E}) \wedge \neg \mathsf{abt}(\mathcal{E}) \\ \Longrightarrow \mathsf{prog\text{-}p}(\mathcal{E}) \,.$$

- If $|\mathcal{E}| \neq \omega$, we know prog-p($\mathcal{E}$) must hold.
- If $|\mathcal{E}| = \omega$, suppose prog-p($\mathcal{E}$) does not hold. Then there exist i and e such that $e \in \mathsf{pend\_inv}(\mathcal{E}(1..i))$ and $\forall j.\ j > i \implies \neg\mathsf{is\_ret}(\mathcal{E}(j))$.
  Suppose the execution generating such $\mathcal{E}$ is:
  $$(W_0, \mathcal{S}_0) \xrightarrow{\mathcal{E}(1)} (W_1, \mathcal{S}_1) \xrightarrow{\mathcal{E}(2)} (W_2, \mathcal{S}_2) \xrightarrow{\mathcal{E}(3)} \dots$$
  Here we write W_0 for $\mathbf{let}\ \mathsf{wr}_1(\Gamma)\ \mathbf{in}\ C_1 \parallel \dots \parallel C_n$ and $\mathcal{S}_0$ for $(\sigma_c, \sigma_a, \circledcirc)$.
  Suppose $\mathsf{tid}(e) = \mathsf{t}$ and $e = \mathcal{E}(i_0)$. We know $\neg\exists j.\ (j > i_0 \wedge \mathsf{match}(e, \mathcal{E}(j)))$.
  Since $\mathsf{fair}(\mathcal{E})$, we know $|(\mathcal{E}|_\mathsf{t})| = \omega$. From the operational semantics, we know there exist $i_1, i_2, \dots$ (an infinite number) such that $i_0 < i_1 < i_2 < \dots$, $\mathcal{E}(i_1) = \mathcal{E}(i_2) = \dots = (\mathsf{t}, \mathbf{obj})$, and the code of thread t in $W_{i_1-1}, W_{i_1}, W_{i_2-1}, W_{i_2}, \dots$ 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 $\mathsf{t}_1, \mathsf{t}_2, \dots$ in the states $\mathcal{S}_{i_1}, \mathcal{S}_{i_2}, \dots$. We know $\mathsf{t}_1 \neq 0$, $\mathsf{t}_2 \neq 0$, ... Since there are only n threads, we know there exist $j, j_1, j_2, \dots$ (an infinite number) such that $\mathsf{t}_j = \mathsf{t}_{j_1} = \mathsf{t}_{j_2} = \dots$. In other words, thread t_j holds the lock infinitely often in the execution.
  Since $\mathsf{fair}(\mathcal{E})$, we know $|(\mathcal{E}|_{\mathsf{t}_j})| = \omega$. By the operational semantics, we know t_j must return infinitely often.
  So we get a contradiction. Thus prog-p($\mathcal{E}$) holds.

Thus we are done. □

Proofs of (B.18). The key is to show the following (B.25).

For any $n, C_1, \dots, C_n, \sigma_c, \sigma_o, \sigma_1$ and σ_a such that $(\sigma_o, \sigma_1) \models P$ and $\sigma_a = \sigma_1 \uplus \{\mathtt{l} \rightsquigarrow 0\}$,
if $(\lfloor \mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \parallel \dots \parallel C_n \rfloor, (\sigma_c, \sigma_o, \circledcirc)) \xrightarrow{\mathcal{E}}{}^{\omega} \cdot$ and $\mathsf{fair}(\mathcal{E})$, then there exists $\mathcal{E}_b$ such that
$(\lfloor \mathbf{let}\ \mathsf{wr}_1(\Gamma)\ \mathbf{in}\ C_1 \parallel \dots \parallel C_n \rfloor, (\sigma_c, \sigma_a, \circledcirc)) \xrightarrow{\mathcal{E}_b}{}^{\omega} \cdot$, $\mathsf{fair}(\mathcal{E}_b)$ and $\mathsf{get\_obsv}(\mathcal{E}) = \mathsf{get\_obsv}(\mathcal{E}_b)$. (B.25)

Following the proof of (B.19), we get:

For any $n, C_1, \dots, C_n, \sigma_c, \sigma_o$ and σ_1 such that $(\sigma_o, \sigma_1) \models P$,
if $(\lfloor \mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \parallel \dots \parallel C_n \rfloor, (\sigma_c, \sigma_o, \circledcirc)) \xrightarrow{\mathcal{E}}{}^{\omega} \cdot$ and $\mathsf{fair}(\mathcal{E})$, then there exists $\mathcal{E}_a$ such that
$(\lfloor \mathbf{let}\ \Gamma\ \mathbf{in}\ C_1 \parallel \dots \parallel C_n \rfloor, (\sigma_c, \sigma_1, \circledcirc)) \xrightarrow{\mathcal{E}_a}{}^{\omega} \cdot$ and $\mathcal{E}\backslash(\_, \mathbf{obj}) = \mathcal{E}_a\backslash(\_, \mathbf{obj})$. (B.26)

Since $\mathsf{fair}(\mathcal{E})$, we can prove that $\mathcal{E}_a$ satisfies the following cltfair in Definition B.63. Since $\mathcal{E}\backslash(\_, \mathbf{obj}) = \mathcal{E}_a\backslash(\_, \mathbf{obj})$, from prog-p($\mathcal{E}$), we know prog-p($\mathcal{E}_a$) holds. By the following Lemma B.64, we can construct the execution of $\mathbf{let}\ \mathsf{wr}_1(\Gamma)\ \mathbf{in}\ C_1 \parallel \dots \parallel C_n$ generating the trace $\mathcal{E}_b$ such that $\mathsf{fair}(\mathcal{E}_b)$ and $\mathcal{E}_a\backslash(\_, \mathbf{obj}) = \mathcal{E}_b\backslash(\_, \mathbf{obj})$ hold. Thus we are done.

Definition B.63. $\mathsf{cltfair}(\mathcal{E}_a)$ iff
$$|\mathcal{E}_a| = \omega \implies \forall \mathsf{t} \in [1..\mathsf{tnum}(\mathcal{E}_a)].\ |(\mathcal{E}_a|_\mathsf{t})| = \omega \vee \mathsf{last}(\mathcal{E}_a|_\mathsf{t}) = (\mathsf{t}, \mathbf{term}) \vee \mathsf{last}(\mathcal{E}_a|_\mathsf{t}) = (\mathsf{t}, \mathbf{obj}) \vee \exists f, n.\ \mathsf{last}(\mathcal{E}_a|_\mathsf{t}) = (\mathsf{t}, f, n)$$

Lemma B.64. *If* $(\lfloor \mathbf{let}\ \Gamma\ \mathbf{in}\ C_1 \parallel \dots \parallel C_n \rfloor, (\sigma_c, \sigma_a, \circledcirc)) \xrightarrow{\mathcal{E}_a}{}^{\omega} \cdot$, $\mathsf{cltfair}(\mathcal{E}_a)$ *and* $\mathsf{prog\text{-}p}(\mathcal{E}_a)$, *then there exists* $\mathcal{E}_b$ *such that* $(\lfloor \mathbf{let}\ \mathsf{wr}_1(\Gamma)\ \mathbf{in}\ C_1 \parallel \dots \parallel C_n \rfloor, (\sigma_c, \sigma_a \uplus \{l \rightsquigarrow 0\}, \circledcirc)) \xrightarrow{\mathcal{E}_b}{}^{\omega} \cdot$, $\mathsf{fair}(\mathcal{E}_b)$ *and* $\mathcal{E}_a\backslash(\_, \mathbf{obj}) = \mathcal{E}_b\backslash(\_, \mathbf{obj})$.

Proof. We construct a simulation relation $\lesssim$ that satisfies the following property.

$$\begin{array}{l}\text{If } \mathcal{E} \models (W_a, \mathcal{S}_a) \lesssim (W_b, \mathcal{S}_b) \text{ and } (W_a, \mathcal{S}_a) \xmapsto{e_a} (W_a', \mathcal{S}_a'), \text{ then}\\ \text{there exist } \mathcal{E}_b, W_b' \text{ and } \mathcal{S}_b' \text{ such that } (W_b, \mathcal{S}_b) \xmapsto{\mathcal{E}_b}{}^{+} (W_b', \mathcal{S}_b'),\\ \mathcal{E} :: e_a \models (W_a', \mathcal{S}_a') \lesssim (W_b', \mathcal{S}_b'), \quad e_a\backslash(\_, \mathbf{obj}) = \mathcal{E}_b\backslash(\_, \mathbf{obj}) \text{ and}\\ \text{if is\_ret}(e_a) \text{ holds, then } \forall e.\ e \in \text{pend\_inv}(\mathcal{E}) \implies \exists i.\ \text{tid}(e) = \text{tid}(\mathcal{E}_b(i)).\end{array} \tag{B.27}$$

The simulation is established as follows.

$$\begin{array}{l}\mathcal{E} \models (\mathbf{let}\ \Gamma\ \mathbf{in}\ C_1 \parallel \ldots \parallel C_n, (\sigma_c, \sigma_a, \{1 \rightsquigarrow \kappa_1, \ldots, n \rightsquigarrow \kappa_n\}))\\ \quad \lesssim (\mathbf{let}\ \mathsf{wr}_1(\Gamma)\ \mathbf{in}\ C_1' \parallel \ldots \parallel C_n', (\sigma_c, \sigma_a', \{1 \rightsquigarrow \kappa_1', \ldots, n \rightsquigarrow \kappa_n'\}))\\ \qquad\qquad \text{if } \forall i.\ (C_i, \sigma_a, \kappa_i) \lesssim_i (C_i', \sigma_a', \kappa_i')\end{array}$$

$$(C, \sigma_a, \circ) \lesssim_{\mathsf{t}} (C, \sigma_a \uplus \{l \rightsquigarrow 0\}, \circ)$$

$$(\langle C\rangle; \mathbf{return}\ E; \mathbf{noret}, \sigma_a, \kappa) \lesssim_{\mathsf{t}} (\mathsf{wr}_1(\langle C\rangle); \mathsf{wr}_1(\mathbf{return}\ E); \mathbf{noret}, \sigma_a \uplus \{1 \rightsquigarrow 0\}, \kappa')$$

$$(\mathbf{return}\ E; \mathbf{noret}, \sigma_a, \kappa) \lesssim_{\mathsf{t}} (\mathsf{wr}_1(\mathbf{return}\ E); \mathbf{noret}, \sigma_a \uplus \{1 \rightsquigarrow 0\}, \kappa')$$

$$\text{where } \kappa' = (s_l \uplus \{\mathsf{u1} \rightsquigarrow \_, \mathsf{u2} \rightsquigarrow \_\}, x, C')\ ,\ \text{if } \kappa = (s_l, x, C')$$

We can prove (B.27) 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 is_ret(e_a) holds, the pending threads in $\mathcal{E}$ must have been executed in the execution of $(W_b, \mathcal{S}_b)$.

With (B.27), we can prove the following by co-induction over the event trace $\mathcal{E}_a$ generated in $(W_a, \mathcal{S}_a) \xmapsto{\mathcal{E}_a}{}^{\omega} \cdot$.

$$\begin{array}{l}\text{If } \epsilon \models (W_a, \mathcal{S}_a) \lesssim (W_b, \mathcal{S}_b),\ (W_a, \mathcal{S}_a) \xmapsto{\mathcal{E}_a}{}^{\omega} \cdot \text{ and prog-p}(\mathcal{E}_a), \text{ then}\\ \text{there exists } \mathcal{E}_b \text{ such that } (W_b, \mathcal{S}_b) \xmapsto{\mathcal{E}_b}{}^{\omega} \cdot,\ \mathcal{E}_a\backslash(\_, \mathbf{obj}) = \mathcal{E}_b\backslash(\_, \mathbf{obj}) \text{ and}\\ \forall e.\ e \in \text{pend\_inv}(\mathcal{E}_b) \implies \forall i.\ \exists j > i.\ \text{tid}(\mathcal{E}_b(j)) = \text{tid}(e).\end{array} \tag{B.28}$$

Since

$$\epsilon \models (\lfloor \mathbf{let}\ \Gamma\ \mathbf{in}\ C_1 \parallel \ldots \parallel C_n \rfloor, (\sigma_c, \sigma_a, \circledcirc)) \lesssim (\lfloor \mathbf{let}\ \mathsf{wr}_1(\Gamma)\ \mathbf{in}\ C_1 \parallel \ldots \parallel C_n \rfloor, (\sigma_c, \sigma_a \uplus \{1 \rightsquigarrow 0\}, \circledcirc))\,,$$

from (B.28), we know there exists $\mathcal{E}_b$ such that

$$(\lfloor \mathbf{let}\ \mathsf{wr}_1(\Gamma)\ \mathbf{in}\ C_1 \parallel \ldots \parallel C_n \rfloor, (\sigma_c, \sigma_a \uplus \{1 \rightsquigarrow 0\}, \circledcirc)) \xmapsto{\mathcal{E}_b}{}^{\omega} \cdot,$$

$\mathcal{E}_a\backslash(\_, \mathbf{obj}) = \mathcal{E}_b\backslash(\_, \mathbf{obj})$ and $\forall e.\ e \in \text{pend\_inv}(\mathcal{E}_b) \implies \forall i.\ \exists j > i.\ \text{tid}(\mathcal{E}_b(j)) = \text{tid}(e)$.
Since $(\lfloor \mathbf{let}\ \Gamma\ \mathbf{in}\ C_1 \parallel \ldots \parallel C_n \rfloor, (\sigma_c, \sigma_a, \circledcirc)) \xmapsto{\mathcal{E}_a}{}^{\omega} \cdot$ and cltfair$(\mathcal{E}_a)$, we know fair$(\mathcal{E}_b)$ holds. Thus we are done. □

C EXAMPLES

In this section, we show the examples that we have verified. The examples cover concurrent list algorithms such as lock coupling list, optimistic list and lazy list, and also the nested locking object and transfer lists.

C.1 Nested locking object with TAS lock

Below we verify the nested locking object with test-and-set lock. Fig. 31 shows the implementation and the auxiliary code for the object. The abstract specification for the object's operation is **skip**. The object is deadlock-free because test-and-set lock allows one to acquire the lock and delay others.

Fig. 32 shows the assertions and specifications. Most of them are the same as that in Sec. 5. Now we use LRG, so we define I to fence over rely and guarantee transitions. Each thread may acquire the lock and release the lock. Also when a thread executes `AB()` or `BA()`, it needs to set the auxiliary string `str` correspondingly. The `str` distinguishes the context and allows to assign layers dynamically.

```
A(){
 1  lock A;
 2  unlock A;
}

B(){
 3  lock B;
 4  unlock B;
}
```

```
AB(){
 5  lock L;
 6  str="AB";
 7  lock A;
 8  lock B;
 9  str="";
10  unlock L;
11  unlock A;
12  unlock B;
}
```

```
BA(){
13  lock L;
14  str="BA";
15  lock B;
16  lock A;
17  str="";
18  unlock L;
19  unlock B;
20  unlock A;
}
```

```
lock X {
21  local b := false;
22  while(!b) {
23    b := cas(&X, 0, t);
24  }
}

unlock X {
25  X := 0;
}
```

Fig. 31. Nested locking object with auxilary code.

The definite actions describe the lock release action. The layer feature is either a variable referring to the lock or a special $\perp$ for $\mathcal{B}_0$. $\mathcal{B}_0$ can never be enabled. It serves as a place holder when no definite action is needed for a command, (e.g., for an assignment command which always terminates). We assign definite action on `L` the highest layer 3, and give $\mathcal{B}_0$ the lowest layer 0. Definite actions for `A` and `B` are normally layer 1, but may also be lifted to layer 2 to allow the dependency among definite actions.

Fig. 33 shows the proofs of `AB()`. The precondition says the thread does not hold any lock. Before verifying each lock statement, we first apply the CSQ rule to strengthen the guarantee conditions to G'. Then we apply the LIVE-FRM rule to pick the needed definite actions for proving the command. Fig. 34 shows the verification of each lock statement. Here we sketch a proof template for lock statement. P is the frame part of memory. The specification of lock X only needs the definite action on X and a place holder $\mathcal{B}_0$ when no definite action is needed. Before entering the while loop, we first apply the HIDE-$\Diamond$ rule to get a $\Diamond$-token. The while rule needs to prove the definite progress condition. When we prove the loop body, we can further apply the LIVE-FRM to allow more local proofs. The proof of test-and-set lock is similar to the proof of it in other objects that will be introduced later. Therefore, we may omit their proofs of lock statement.

$$I \stackrel{\text{def}}{=} \mathsf{lock(L)} * \mathsf{lock(A)} * \mathsf{lock(B)} * \mathsf{str=\_}$$

$$\mathsf{lock}_s(\mathsf{x}) \stackrel{\text{def}}{=} (\mathsf{x} = s)$$

$$\mathsf{lock(x)} \stackrel{\text{def}}{=} \exists s.\, \mathsf{lock}_s(\mathsf{x})$$

$$\mathsf{unlocked(x)} \stackrel{\text{def}}{=} \mathsf{lock}_0(\mathsf{x})$$

$$\mathsf{locked(x)} \stackrel{\text{def}}{=} \exists \mathsf{t}.\, \mathsf{lockedBy}_\mathsf{t}(\mathsf{x})$$

$$\mathsf{lockedBy}_\mathsf{t}(\mathsf{x}) \stackrel{\text{def}}{=} \mathsf{lock}_\mathsf{t}(\mathsf{x}) \wedge (\mathsf{t} \neq 0)$$

$$\mathsf{notOwn}_\mathsf{t}(\mathsf{x}) \stackrel{\text{def}}{=} \exists \mathsf{t}'.\, \mathsf{lock}_{\mathsf{t}'}(\mathsf{x}) \wedge (\mathsf{t} \neq \mathsf{t}')$$

$$R_\mathsf{t} \stackrel{\text{def}}{=} \bigvee_{\mathsf{t}' \neq \mathsf{t}} G_{\mathsf{t}'}$$

$$G_\mathsf{t} \stackrel{\text{def}}{=} (G_\mathsf{t}(\mathsf{L}) \vee G_\mathsf{t}(\mathsf{A}) \vee G_\mathsf{t}(\mathsf{B}) \vee \mathit{SetStr}_\mathsf{t} \vee \mathsf{Id}) * \mathsf{Id} \wedge (I \ltimes I)$$

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

$$\mathit{Acq}_\mathsf{t}(x) \stackrel{\text{def}}{=} \mathsf{unlocked(x)} \ltimes_1 \mathsf{lockedBy}_\mathsf{t}(\mathsf{x})$$

$$\mathit{Rel}_\mathsf{t}(x) \stackrel{\text{def}}{=} \mathsf{lockedBy}_\mathsf{t}(\mathsf{x}) \ltimes_0 \mathsf{unlocked(x)}$$

$$\mathit{SetStr}_\mathsf{t} \stackrel{\text{def}}{=} (\mathsf{lockedBy}_\mathsf{t}(\mathsf{L}) * \mathsf{str=\_}) \ltimes_0 (\mathsf{lockedBy}_\mathsf{t}(\mathsf{L}) * \mathsf{str=\_})$$

$$\mathcal{D}_\mathsf{t} \stackrel{\text{def}}{=} \mathcal{B}_\mathsf{t}(\mathsf{L}) \wedge \mathcal{B}_\mathsf{t}(\mathsf{A}) \wedge \mathcal{B}_\mathsf{t}(\mathsf{B}) \wedge \mathcal{B}_0$$

$$\mathit{LayFeat} \stackrel{\text{def}}{=} \mathit{Var} \mid \bot$$

$$\mathcal{B}_\mathsf{t}(\mathsf{x}) \stackrel{\text{def}}{=} (\mathsf{bp}_\mathsf{t}(\mathsf{x}) \rightsquigarrow \mathsf{bq(x)}, \mathsf{x})$$

$$\mathcal{B}_0 \stackrel{\text{def}}{=} (\mathsf{False} \rightsquigarrow \mathsf{True}, \bot)$$

$$\mathsf{bp}_\mathsf{t}(\mathsf{x}) \stackrel{\text{def}}{=} \mathsf{lockedBy}_\mathsf{t}(\mathsf{x}) * \mathbf{true}$$

$$\mathsf{bq(x)} \stackrel{\text{def}}{=} \mathsf{unlocked(x)} * \mathbf{true}$$

$$\mathfrak{L}(\mathfrak{S}, \mathcal{B}) = \begin{cases} 3 & \text{if } \mathcal{B}.\mathcal{F} = \mathsf{L} \\ 2 & \text{if } \mathcal{B}.\mathcal{F} = \mathsf{A} \wedge \mathfrak{S} \models \mathsf{P}_{AB} \\ 2 & \text{if } \mathcal{B}.\mathcal{F} = \mathsf{B} \wedge \mathfrak{S} \models \mathsf{P}_{BA} \\ 0 & \text{if } \mathcal{B}.\mathcal{F} = \bot \\ 1 & \text{otherwise} \end{cases}$$

$$\mathsf{P}_{AB} \stackrel{\text{def}}{=} \mathsf{str="AB"} * \mathbf{true}$$

$$\mathsf{P}_{BA} \stackrel{\text{def}}{=} \mathsf{str="BA"} * \mathbf{true}$$

Fig. 32. Assertions and specifications for the nested locking object with TAS locks.

```
AB()
```
$\mathfrak{L}, \mathcal{D}, R, G, I \vdash$
$\{\mathsf{notOwn_t(L)} * \mathsf{notOwn_t(A)} * \mathsf{notOwn_t(B)} * \mathsf{str=\_} \wedge \blacklozenge(3) \wedge \mathsf{arem}(\mathbf{skip})\}$
// apply CSQ rule and LIVE-FRM rule
```
1  lock L;
```
$\{\mathsf{lockedBy_t(L)} * \mathsf{notOwn_t(A)} * \mathsf{notOwn_t(B)} * \mathsf{str=\_} \wedge \blacklozenge(2) \wedge \mathsf{arem}(\mathbf{skip})\}$
```
2  <str := "AB";>
```
$\{\mathsf{lockedBy_t(L)} * \mathsf{notOwn_t(A)} * \mathsf{notOwn_t(B)} * \mathsf{str="AB"} \wedge \blacklozenge(2) \wedge \mathsf{arem}(\mathbf{skip})\}$
```
3  lock A;
```
$\{\mathsf{lockedBy_t(L)} * \mathsf{lockedBy_t(A)} * \mathsf{notOwn_t(B)} * \mathsf{str="AB"} \wedge \blacklozenge(1) \wedge \mathsf{arem}(\mathbf{skip})\}$
```
4  lock B; <str := "";>
```
$\{\mathsf{lockedBy_t(L)} * \mathsf{lockedBy_t(A)} * \mathsf{lockedBy_t(B)} * \mathsf{str=""} \wedge \mathsf{arem}(\mathbf{skip})\}$
```
5  unlock L; unlock A; unlock B;
```
$\{\mathsf{notOwn_t(L)} * \mathsf{notOwn_t(A)} * \mathsf{notOwn_t(B)} * \mathsf{str=\_} \wedge \mathsf{arem}(\mathbf{skip})\}$

Fig. 33. Proofs for the AB() method of nested locking object.

$$\mathfrak{L}_X(\mathfrak{S},\mathcal{B}) = \begin{cases} 1 & \text{if } \mathcal{B}.\mathcal{F} = X \\ 0 & \text{if } \mathcal{B}.\mathcal{F} = \bot \\ \textit{undefined} & \text{otherwise} \end{cases}$$

$$\mathfrak{L}_0(\mathfrak{S},\mathcal{B}) = \begin{cases} 0 & \text{if } \mathcal{B}.\mathcal{F} = \bot \\ \textit{undefined} & \text{otherwise} \end{cases}$$

$$f(\mathfrak{S},X) = \begin{cases} 1 & \text{if } \mathfrak{S} \models \mathsf{locked}(X) * \mathbf{true} \\ 0 & \text{if } \mathfrak{S} \models \mathsf{unlocked}(X) * \mathbf{true} \end{cases}$$

$$G' \stackrel{\text{def}}{=} (Acq_t(x) * \mathsf{Id}) \wedge (I \ltimes I)$$

$$J \stackrel{\text{def}}{=} \mathsf{lock}(X) * P$$

$$Q \stackrel{\text{def}}{=} \mathsf{unlocked}(X) * P$$

$$\mathcal{D}' \stackrel{\text{def}}{=} \mathcal{B}(X)$$

```
lock X
```
$\mathfrak{L}_X, \mathcal{B}(X) \wedge \mathcal{B}_0, R, G' \vdash$
$\{(\mathsf{notOwn}_t(X) \wedge \blacklozenge) * P \wedge \mathsf{arem}(\mathbf{skip})\}$
```
6  local b := false;
```
// apply HIDE-◇ rule
$\{((\mathsf{notOwn}_t(X) \wedge \blacklozenge \wedge \Diamond \wedge \neg b) \vee (\mathsf{lockedBy}_t(X) \wedge b)) * P \wedge \mathsf{arem}(\mathbf{skip})\}$
// apply WHL rule
```
7  while(!b) {
```
// apply LIVE-FRM rule
$\mathfrak{L}_0, \mathcal{B}_0, R, G' \vdash$
$\{((\mathsf{unlocked}(X) \wedge \blacklozenge) \vee (\mathsf{locked}(X) \wedge \blacklozenge \wedge \Diamond)) * P \wedge \mathsf{arem}(\mathbf{skip})\}$
```
8    b := cas(&X, 0, t); }
```

Fig. 34. Proof reuse of lock X for the nested locking object with TAS locks.

C.2 Nested locking object with ticket lock

Next, we show the nested locking object implemented with ticket locks. Here, we only show the implementation of lock X. We implicitly assume every instruction that accesses the shared state is in an atomic block. We introduce some auxiliary structures to facilitate the proof. As shown in Fig. 35, we introduce the explicit tickets $\texttt{ticket}_0$, $\texttt{ticket}_1$, ... Each $\texttt{ticket}_i$ 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.

```
struct Lock {
  int owner;
  int next;
  int[] ticket; //initially all -1
}

lock X {
 1  local i, o;
 2  <i := getAndInc(X.next); X.ticket_pi := cid>;
 3  <o := X.owner>;
 4  while (i <> o) {
 5    <o := X.owner>;
 6}

unlock X {
 7  <X.owner++>;
 8}
```

Fig. 35. Concrete implementation and auxiliary code for ticket lock.

Fig. 36 defines the precise invariant I which determines the boundaries of shared states. $\text{lock}(L, tl, n_1, n_2)$ contains the auxiliary `tickets` in addition to `owner` and `next`, where tl (hidden in the definition of I) is the list of the threads $\texttt{ticket}_{\texttt{owner}}, \ldots, \texttt{ticket}_{\texttt{next}-1}$. We also use $\text{locked}(L, tl, n_1, n_2)$ 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 $\text{lockIrr}_\text{t}(L, tl, n_1, n_2)$ short for $\text{lock}(L, tl, n_1, n_2) \wedge (\text{t} \notin tl)$. That is, the thread t is irrelevant to the lock: it does not acquire or request the lock.

The guarantee condition G in Fig. 36 defines the atomic actions of a thread t. $\mathit{ReqLock}_\text{t}$ adds the thread t at the end of tl of the threads requesting the lock and increments `next`. $\mathit{RelLock}_\text{t}$ releases the lock.

The definite action requires whenever a thread holds a lock with $\texttt{owner} = n_1$, it should eventually release the lock by incrementing `owner` to $n_1 + 1$. The layers are defined similarly to the objects with TAS locks.

Fig.37 shows the proof outline of lock L. We also assume a frame part that stay unchanged during lock L. Only the stability proof needs to open FRM. And we only need to prove it once. To verify the loop at lines 4-6, we first define J and Q. Both J and Q are stronger than I. For thread t, J_t says t is requesting the lock. Here $\text{tlocked}_{tl_1,\text{t},tl_2}(L, n_1, X, n_2)$ says (1) t takes the ticket number X, which satisfies $n_1 = L.\texttt{owner} \leq X < L.\texttt{next} = n_2$, 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 $tl_1 :: \mathrm{t} :: tl_2$ holds the lock. Q_t specifies the case when tl_1 is empty (thus $X = L.\mathtt{owner}$). The loop terminates when Q holds. We also strengthen the guarantee of the loop to $G' \stackrel{\text{def}}{=} [I]$, the identity transitions. We have $\mathrm{Sta}(\{J, Q\}, G' \vee R)$.

Next we define f and prove $J \Rightarrow (R, G' : \mathcal{D}_L \wedge \mathcal{B}_0 \xrightarrow{f} Q)$. 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 $J \Rightarrow (R, G' : \mathcal{D}_L \wedge \mathcal{B}_0 \xrightarrow{f} Q)$ holds due to the following reasons:

(1) Each action in R or G' either increases $\mathtt{owner}$ or keeps $\mathtt{owner}$ 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.

$tl ::= \epsilon \mid \mathsf{t} :: tl$

$\mathsf{list2set}(\epsilon) \stackrel{\text{def}}{=} \emptyset$

$\mathsf{list2set}(\mathsf{t} :: tl) \stackrel{\text{def}}{=} \{\mathsf{t}\} \cup \mathsf{list2set}(tl)$

$I \stackrel{\text{def}}{=} \mathsf{lock}(L) * (lock)(A) * (lock)(B) * str = \_$

$\mathsf{lock}(L) \stackrel{\text{def}}{=} \exists n_1, n_2.\ \mathsf{lock}(L, n_1, n_2)$

$\mathsf{lock}(L, n_1, n_2) \stackrel{\text{def}}{=} \exists tl.\ \mathsf{lock}(L, tl, n_1, n_2)$

$\mathsf{lock}(L, tl, n_1, n_2) \stackrel{\text{def}}{=} ((L.\mathtt{owner} = n_1) * (L.\mathtt{next} = n_2) \wedge (n_1 \le n_2)) *$
$\quad \mathsf{tickets}(L, 0, n_1) * \mathsf{tickets}(L, tl, n_1, n_2) * \mathsf{tickets\_new}(L, n_2)$

$\mathsf{tickets}(L, n_1, n_2) \stackrel{\text{def}}{=} \exists tl.\ \mathsf{tickets}(L, tl, n_1, n_2)$

$\mathsf{tickets}(L, tl, n_1, n_2) \stackrel{\text{def}}{=}$
$\quad (tl = \epsilon) \wedge (n_1 = n_2) \wedge \mathsf{emp} \ \vee\ \exists \mathsf{t}, tl'.\ (tl = \mathsf{t} :: tl') \wedge (\mathsf{t} \notin \mathsf{list2set}(tl')) \wedge (L.\mathtt{ticket}_{n_1} = \mathsf{t}) * \mathsf{tickets}(tl', n_1 + 1, n_2)$

$\mathsf{tickets\_new}(L, n_2) \stackrel{\text{def}}{=} (\circledast_{i \ge n_2} L.\mathtt{ticket}_i = -1)$

$\mathsf{lock\_irr}_\mathsf{t}(L) \stackrel{\text{def}}{=} \exists n_1, n_2.\ \mathsf{lock\_irr}_\mathsf{t}(L, n_1, n_2)$

$\mathsf{lock\_irr}_\mathsf{t}(L, n_1, n_2) \stackrel{\text{def}}{=} \exists tl.\ \mathsf{lock\_irr}_\mathsf{t}(L, tl, n_1, n_2)$ $\qquad$ $\mathsf{lock\_irr}_\mathsf{t}(L, tl, n_1, n_2) \stackrel{\text{def}}{=} \mathsf{lock}(L, tl, n_1, n_2) \wedge (\mathsf{t} \notin \mathsf{list2set}(tl))$

$R_\mathsf{t} \stackrel{\text{def}}{=} \bigvee_{\mathsf{t}' \neq \mathsf{t}} G_{\mathsf{t}'}$

$G_\mathsf{t} \stackrel{\text{def}}{=} (G_\mathsf{t}(\mathsf{L}) \vee G_\mathsf{t}(\mathsf{A}) \vee G_\mathsf{t}(\mathsf{B}) \vee \mathit{SetStr}_\mathsf{t} \vee \mathsf{Id}) * \mathsf{Id} \wedge (I \ltimes I)$

$G_\mathsf{t}(\mathsf{L}) \stackrel{\text{def}}{=} \mathit{ReqLock}_\mathsf{t}(L) \vee \mathit{RelLock}_\mathsf{t}(L)$

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

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

$\mathcal{D}_\mathsf{t} \stackrel{\text{def}}{=} \mathcal{D}_\mathsf{t}(\mathsf{L}) \wedge \mathcal{D}_\mathsf{t}(\mathsf{A}) \wedge \mathcal{D}_\mathsf{t}(\mathsf{B}) \wedge \mathcal{B}_0$ $\qquad$ $\mathit{LayFeat} \stackrel{\text{def}}{=} \mathit{Var} \mid \bot$

$\mathcal{D}_\mathsf{t}(L) \stackrel{\text{def}}{=} \forall n_1.\ \mathcal{B}_\mathsf{t}(L, n_1)$ $\qquad$ $\mathcal{B}_\mathsf{t}(L, n_1) \stackrel{\text{def}}{=} (\mathsf{bp}_\mathsf{t}(L, n_1) \rightsquigarrow \mathsf{bq}(L, n_1), L)$

$bp_\mathsf{t}(L, n_1) \stackrel{\text{def}}{=} \exists tl, n_2.\ \mathsf{lock}(L, \mathsf{t} :: tl, n_1, n_2) * \mathsf{true} \wedge I$

$bq_\mathsf{t}(L, n_1) \stackrel{\text{def}}{=} \exists n_2.\ \mathsf{lock\_irr}_\mathsf{t}(L, n_1 + 1, n_2) * \mathsf{true} \wedge I$

$$\mathfrak{L}(\mathfrak{S}, \mathcal{B}) = \begin{cases} 3 & \text{if } \mathcal{B}.\mathcal{F} = \mathsf{L} \\ 2 & \text{if } \mathcal{B}.\mathcal{F} = \mathsf{A} \wedge \mathfrak{S} \models \mathrm{str}=\text{"AB"} * \mathbf{true} \\ 2 & \text{if } \mathcal{B}.\mathcal{F} = \mathsf{B} \wedge \mathfrak{S} \models \mathrm{str}=\text{"BA"} * \mathbf{true} \\ 0 & \text{if } \mathcal{B}.\mathcal{F} = \bot \\ 1 & \text{otherwise} \end{cases}$$

Fig. 36. Invariant, rely/guarantee and definite action of nested locking object with ticket lock.

$\mathsf{tlocked}_{L,tl_1,\mathsf{t},tl_2}(n_1,n,n_2) \stackrel{\text{def}}{=}$
$(\mathsf{list2set}(tl_1) \cap \mathsf{list2set}(\mathsf{t}::tl_2) = \emptyset) \wedge ((L.\mathsf{owner} = n_1) * (L.\mathsf{next} = n_2) \wedge (n_1 \le n < n_2))$
$* \mathsf{tickets}(L,0,n_1) * \mathsf{tickets}(L,tl_1,n_1,n) * \mathsf{tickets}(L,\mathsf{t}::tl_2,n,n_2) * \mathsf{tickets\_new}(L,n_2)$

$P_0(L,n_1,n,n_2) \stackrel{\text{def}}{=} \exists tl_1.\ P_0(L,tl_1,n_1,n,n_2)$ $\qquad$ $P_3(L,n_1,n,n_2) \stackrel{\text{def}}{=} \exists \mathsf{t}', tl_1.\ P_0(L,\mathsf{t}'::tl_1,n_1,n,n_2)$

$P_0(L,tl_1,n_1,n,n_2) \stackrel{\text{def}}{=} \exists tl_2.\ \mathsf{tlocked}_{tl_1,\mathsf{t},tl_2}(L,n_1,n,n_2)$

$P_1(n_1,n_2) \stackrel{\text{def}}{=} \exists tl.\ \mathsf{lock}(L,\mathsf{t}::tl,n_1,n_2)$

$P_2(n_1,n_2) \stackrel{\text{def}}{=} \exists tl.\ \mathsf{lock}(L,\mathsf{t}::tl,n_1,n_2)$

$J \stackrel{\text{def}}{=} \exists n_1,n_2.\ P_0(L,n_1,n,n_2) * \mathsf{FRM}$

$Q \stackrel{\text{def}}{=} \exists tl,n_1,n_2.\ \mathsf{lock}(L,\mathsf{t}::tl,n_1,n_2) * \mathsf{FRM}$

$G_L \stackrel{\text{def}}{=} (\mathit{ReqLock}_{\mathsf{t}}(L) \vee \mathsf{Id}) * \mathsf{Id} \wedge (I \ltimes I)$

$G' \stackrel{\text{def}}{=} [I]$

$f(\mathfrak{S}) = k$ iff $\mathfrak{S} \models (n - \mathsf{owner} = k)$

$$\mathfrak{L}_L(\mathfrak{S},\mathcal{B}) = \begin{cases} 1 & \text{if } \mathcal{B}.\mathcal{F} = L \\ 0 & \text{if } \mathcal{B}.\mathcal{F} = \bot \\ \mathit{undefined} & \text{otherwise} \end{cases}$$

$$\mathfrak{L}_0(\mathfrak{S},\mathcal{B}) = \begin{cases} 0 & \text{if } \mathcal{B}.\mathcal{F} = \bot \\ \mathit{undefined} & \text{otherwise} \end{cases}$$

```
lock L:
 1  local i, o;
```
$\mathfrak{L}_L, \mathcal{D}(L) \wedge \mathcal{B}_0, R, G_L \vdash$
$\{\exists n_1,n_2.\ \mathsf{lock\_irr}_{\mathsf{t}}(L,n_1,n_2) * \mathsf{FRM} \wedge \mathsf{arem}(\mathsf{Aop})\}$
```
 2  < i := getAndInc(L.next);  L.ticket_i := cid;>
```
$\{\exists n_1,n,n_2.\ (P_0(L,n_1,n,n_2) \wedge (\mathtt{i} = n)) * \mathsf{FRM} \wedge \mathsf{arem}(\mathsf{Aop})\}$
$\{\exists n_1,n_2.\ (P_0(L,n_1,n,n_2) \wedge (\mathtt{i} = n)) * \mathsf{FRM} \wedge \mathsf{arem}(\mathsf{Aop})\}$
```
 3  o := L.owner;
```
$\{\exists n_1,n_2.\ (P_0(L,n_1,n,n_2) \wedge (\mathtt{i} = n) \wedge (\mathtt{o} \le n_1)) * \mathsf{FRM} \wedge \mathsf{arem}(\mathsf{Aop})\}$
$\{\exists n_1,n_2.\ (P_0(L,n_1,n,n_2) \wedge (\mathtt{i} = n) \wedge (\mathtt{o} \le n_1)) * \mathsf{FRM} \wedge \mathsf{arem}(\mathsf{Aop}) \wedge \Diamond(n - \mathtt{o})\}$
```
 4  while (i != o) {
```
$\mathfrak{L}_0, \mathcal{B}_0, R, G' \vdash$
$\{\exists n_1,n_2.\ (P_1(L,n,n_2) \wedge \Diamond(n - (\mathtt{o}+1)) \vee P_3(L,n_1,n,n_2) \wedge \Diamond(n - \mathtt{o})) \wedge (\mathtt{o} \le n_1 \le n = \mathtt{i}) \wedge (\mathtt{o} \ne \mathtt{i}) \wedge \mathsf{arem}(\mathsf{Aop})\}$
```
 5    o := L.owner;
```
$\{\exists n_1,n_2.\ P_0(L.n_1,n,n_2) \wedge (\mathtt{i} = n) \wedge (\mathtt{o} \le n_1) \wedge \mathsf{arem}(\mathsf{Aop}) \wedge \Diamond(n - \mathtt{o})\}$
```
 6  }
```

Fig. 37. Proof reuse of ticket locks.

C.3 Lock-coupling list with ticket lock

Fig. 38 and 39 shows 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.

```
ADD(E) {
  local R;
  < if (E ∈ S) {
      R := false;
    } else {
      S := S ∪ {E};  R := true;
    }
  >
  return R;
}
```

```
RMV(E) {
  local R;
  < if (E ∉ S) {
      R := false;
    } else {
      S := S \ {E};  R := true;
    }
  >
  return R;
}
```

(a) Abstract operations.

```
bool[] toadd; //initially all false

struct Lock {
  int lowner;
  int lnext;
  int[] ticket; //initially all -1
}
struct Node {
  struct Lock lock;
  int data;
  struct Node *next;
}
```

```
struct List {
  struct Node *Head;
}

initialize(){
  Head := cons(0, 0, MIN, null);
  Head.next := cons(0, 0, MAX, null);
}
```

(b) Data structures.

Fig. 38. Lock-coupling list.

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 nested locking object with ticket locks). Also, we introduce the auxiliary toadd bit for each thread to indicate whether the thread attempts to do the add operation. $\texttt{toadd}_t$ 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

```
add(e) {
 1  local p, c, x, pi;
 2  local po, ci, co, u, r, aux;
 3  <p := Head; toadd_cid := true>;
 4  lock(p);
 5  c := p.next;
 6  <u := c.data;
 7  aux := metric(p)>;
 8  while (u < e) {
 9    lock(c);
10    unlock(p);
11    p := c;
12    c := p.next;
13    <u := c.data;
14    aux := metric(p)>;
15  }
16  if (u != e) {
17    x := cons(0, 0, e, c);
18    <p.next := x;
19    toadd_cid := false>;
20    r := true;
21  } else {
22    <r := false;
23    toadd_cid := false>;
24  }
25  unlock(p);
26  return r;
}
```

```
rmv(e) {
 1  local p, c, n, pi;
 2  po, ci, co, u, r, aux;
 3  p := Head;
 4  lock(p);
 5  c := p.next;
 6  <u := c.data;
 7  aux := metric(p)>;
 8  while (u < e) {
 9    lock(c);
10    unlock(p);
11    p := c;
12    c := p.next;
13    <u := c.data;
14  aux := metric(p)>;
15  }
16  if (u = e) {
17    lock(c);
18    n := c.next;
19    p.next := n;
20    unlock(p);
21    dispose(c);
22    r := true;
23  } else {
24    unlock(p);
25    r := false;
26  }
27  return r;
}
```

(c) Concrete implementations for add and rmv.

Fig. 39. Lock-coupling list.

metric for the loop of list traversal. Fig. 40 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. 41 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. 42 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

```
int metric(p){
 1  local n, l, s, o, t;
 2  n := p.next; l := 0; s := ∅;
 3  while (n <> null) {
 4    l++;
 5    o := n.lowner;
 6    while (o < n.lnext) {
 7      t := n.ticket_o;
 8      if (toadd_t = true) s := s ∪ {t};
 9    }
10    n := n.next;
11  }
12  l := l + |s|;
13  return l;
}
```

$$
\begin{aligned}
|\emptyset| &\stackrel{\text{def}}{=} 0 \\
|S \cup \{x\}| &\stackrel{\text{def}}{=} |S| + 1 \\
\\
\mathsf{len}(\epsilon) &\stackrel{\text{def}}{=} 0 \\
\mathsf{len}(v :: A) &\stackrel{\text{def}}{=} \mathsf{len}(A) + 1 \\
\\
\mathsf{toaddThrds}(S) &\stackrel{\text{def}}{=} \{\mathsf{t} \mid (\mathsf{t} \in S) \wedge (\mathsf{toadd}_\mathsf{t} = \mathsf{true})\}
\end{aligned}
$$

Fig. 40. Auxiliary code to compute the metric for the list traversal.

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.

We can modularly specify the layered definite actions for lock coupling lists. Each thread should ensure to release the lock on a node, or the lock is disposed due to the `rmv` method. Here, we use the data on the node as its layer, since the definite actions at list head should have higher layer. We define a total order on *Layer* reversely. If the value of *Layer* is smaller, then its layer is higher. Using the data value is not the only way to define the layer. We also could assign layer by calculating the node's position in the list. With the LIVE-FRM rule, the verification for each lock only specifies locally-needed definite actions and layer function so the progress proofs can be reused. We can sketch a proof template for each lock statement, which is very similar to the verification of the ticket lock for nested locking object and omitted here.

The whole proof follows the basic linearizability proof of lock-coupling list (e.g., see [Liang and Feng 2013]). Now we also verify the starvation-freedom of the object.

$$A ::= \epsilon \mid v :: A \qquad tl ::= \epsilon \mid \mathsf{t} :: tl \qquad L ::= \epsilon \mid tl{+}L \qquad B \in \mathit{ThrdID} \rightharpoonup \mathit{Bool}$$

$$I \stackrel{\text{def}}{=} \exists A, L.\ \mathsf{toadds} * \mathsf{ls}_L(\mathsf{Head}, A, \mathtt{null}) * \mathsf{ss}(A)$$

$$P_\mathsf{t} \stackrel{\text{def}}{=} \exists A, L.\ \mathsf{toadd}_\mathsf{t}(\mathsf{false}) * \mathsf{ls\_irr}_{\mathsf{t},L}(\mathsf{Head}, A, \mathtt{null}) * \mathsf{ss}(A)$$

$$\mathsf{toadds} \stackrel{\text{def}}{=} \exists B.\ \mathsf{toadds}(B) \qquad \mathsf{toadds}(B) \stackrel{\text{def}}{=} (\circledast_\mathsf{t} \mathsf{toadd}_\mathsf{t} = B(\mathsf{t}))$$

$$\mathsf{toadd}_\mathsf{t}(b) \stackrel{\text{def}}{=} \exists B.\ \mathsf{toadd}_\mathsf{t}(b, B) \qquad \mathsf{toadd}_\mathsf{t}(b, B) \stackrel{\text{def}}{=} (\mathsf{toadd}_\mathsf{t} = b) * (\circledast_{\mathsf{t}' \neq \mathsf{t}} \mathsf{toadd}_{\mathsf{t}'} = B(\mathsf{t}'))$$

$$\begin{aligned}&\mathsf{ls}_L(x, A, y) \stackrel{\text{def}}{=}\\ &\quad (x = y \wedge A = \epsilon \wedge L = \epsilon \wedge \mathsf{emp})\\ &\quad \vee (x \neq y \wedge \exists z, v, A', tl, L'.\ A = v :: A' \wedge L = tl{+}L' \wedge \mathsf{N}_{tl}(x, v, z) * \mathsf{ls}_{L'}(z, A', y))\end{aligned}$$

$$\begin{aligned}&\mathsf{ls\_irr}_{\mathsf{t},L}(x, A, y) \stackrel{\text{def}}{=}\\ &\quad (x = y \wedge A = \epsilon \wedge L = \epsilon \wedge \mathsf{emp})\\ &\quad \vee (x \neq y \wedge \exists z, v, A', tl, L'.\ A = v :: A' \wedge L = tl{+}L' \wedge \mathsf{N\_irr}_{\mathsf{t},tl}(x, v, z) * \mathsf{ls\_irr}_{\mathsf{t},L'}(z, A', y))\end{aligned}$$

$$\begin{aligned}&\mathsf{ls\_unlocked}(x, A, y) \stackrel{\text{def}}{=}\\ &\quad (x = y \wedge A = \epsilon \wedge \mathsf{emp}) \ \vee \ (x \neq y \wedge \exists z, v, A'.\ A = v :: A' \wedge \mathsf{U}(x, v, z) * \mathsf{ls\_unlocked}(z, A', y))\end{aligned}$$

$$\mathsf{N}_{tl}(p, v, y) \stackrel{\text{def}}{=} \mathsf{lock}_{tl}(p) * (p.\mathtt{data} = v) * (p.\mathtt{next} = y)$$

$$\mathsf{N\_irr}_{\mathsf{t},tl}(p, v, y) \stackrel{\text{def}}{=} \exists n.\ \mathsf{N\_irr}_{\mathsf{t},tl;n}(p, v, y) \qquad \mathsf{L}_{tl}(p, v, y) \stackrel{\text{def}}{=} \exists n.\ \mathsf{L}_{tl;n}(p, v, y)$$

$$\mathsf{N\_irr}_{\mathsf{t},tl;n}(p, v, y) \stackrel{\text{def}}{=} \mathsf{lock\_irr}_{\mathsf{t},tl;n}(p) * (p.\mathtt{data} = v) * (p.\mathtt{next} = y)$$

$$\mathsf{L}_{tl;n_1}(p, v, y) \stackrel{\text{def}}{=} \mathsf{locked}_{tl;n_1}(p) * (p.\mathtt{data} = v) * (p.\mathtt{next} = y)$$

$$\mathsf{U}(p, v, y) \stackrel{\text{def}}{=} \mathsf{unlocked}(p) * (p.\mathtt{data} = v) * (p.\mathtt{next} = y)$$

$$\mathsf{lock}_{tl}(p) \stackrel{\text{def}}{=} \exists n.\ \mathsf{lock}_{tl;n}(p) \qquad \mathsf{lock}_{tl;n}(p) \stackrel{\text{def}}{=} \mathsf{locked}_{tl;n}(p) \vee (tl = \epsilon \wedge \mathsf{unlocked}_n(p))$$

$$\mathsf{lock\_irr}_{\mathsf{t},tl}(p) \stackrel{\text{def}}{=} \exists n.\ \mathsf{lock\_irr}_{\mathsf{t},tl;n}(p) \qquad \mathsf{lock\_irr}_{\mathsf{t},tl;n}(p) \stackrel{\text{def}}{=} \mathsf{lock}_{tl;n}(p) \wedge (\mathsf{t} \notin tl)$$

$$\mathsf{locked}_{tl}(p) \stackrel{\text{def}}{=} \exists n.\ \mathsf{locked}_{tl;n}(p) \qquad \mathsf{unlocked}(p) \stackrel{\text{def}}{=} \exists n.\ \mathsf{unlocked}_n(p)$$

$$\begin{aligned}&\mathsf{locked}_{tl;n_1}(p) \stackrel{\text{def}}{=}\\ &\quad \exists \mathsf{t}, tl', n_2.\ (tl = \mathsf{t} :: tl') \wedge (\mathsf{t} \notin tl') \wedge ((p.\mathtt{lowner} = n_1) * (p.\mathtt{lnext} = n_2) \wedge (n_1 < n_2))\\ &\quad * \mathsf{tickets}(p, 0, n_1) * \mathsf{tickets}_{tl}(p, n_1, n_2) * \mathsf{tickets\_new}(p, n_2)\end{aligned}$$

$$\mathsf{unlocked}_n(p) \stackrel{\text{def}}{=} (p.\mathtt{lowner} = n) * (p.\mathtt{lnext} = n) * \mathsf{tickets}(p, 0, n) * \mathsf{tickets\_new}(p, n)$$

$$\mathsf{tickets}(p, n_1, n_2) \stackrel{\text{def}}{=} \exists tl.\ \mathsf{tickets}_{tl}(p, n_1, n_2)$$

$$\mathsf{tickets}_{tl}(p, n_1, n_2) \stackrel{\text{def}}{=} \begin{cases} (n_1 = n_2) \wedge \mathsf{emp} & \text{if } tl = \epsilon \\ (p.\mathtt{ticket}_{n_1} = \mathsf{t}) * \mathsf{tickets}_{tl'}(p, n_1 + 1, n_2) & \text{if } tl = \mathsf{t} :: tl' \end{cases}$$

$$\mathsf{tickets\_new}(p, n_2) \stackrel{\text{def}}{=} (\circledast_{i \geq n_2} p.\mathtt{ticket}_i = -1)$$

$$\mathsf{s}(A) \stackrel{\text{def}}{=} \exists A'.\ (A = \mathtt{MIN} :: A' :: \mathtt{MAX}) \wedge \mathsf{sorted}(A)$$

$$\mathsf{ss}(A) \stackrel{\text{def}}{=} \exists A'.\ (A = \mathtt{MIN} :: A' :: \mathtt{MAX}) \wedge \mathsf{sorted}(A) \wedge (\mathsf{S} = A')$$

$$\mathsf{sorted}(A) \stackrel{\text{def}}{=} \begin{cases} \mathsf{true} & \text{if } A = \epsilon \vee A = v :: \epsilon \\ (v_1 < v_2) \wedge \mathsf{sorted}(v_2 :: A') & \text{if } A = v_1 :: v_2 :: A' \end{cases}$$

$$\mathsf{t} \in tl \ \text{ iff } \ \exists \mathsf{t}', tl'.\ (tl = \mathsf{t}' :: tl') \wedge (\mathsf{t} = \mathsf{t}' \vee \mathsf{t} \in tl')$$

Fig. 41. Invariant and precondition of the lock-coupling list.

$R_\mathsf{t} \stackrel{\text{def}}{=} \bigvee_{\mathsf{t}' \neq \mathsf{t}} G_{\mathsf{t}'}$

$G_\mathsf{t} \stackrel{\text{def}}{=} (\mathit{ToAdd}_\mathsf{t} \vee \mathit{Add}_\mathsf{t} \vee \mathit{FailedAdd}_\mathsf{t} \vee \mathit{Rmv}_\mathsf{t} \vee \mathit{ReqLockH}_\mathsf{t} \vee \mathit{ReqLock2}_\mathsf{t} \vee \mathit{Unlock}_\mathsf{t} \vee \mathsf{Id}) * \mathsf{Id} \wedge (I \ltimes I)$

$\mathit{ToAdd}_\mathsf{t} \stackrel{\text{def}}{=} \exists A, L.\ [\mathsf{ls\_irr}_{\mathsf{t},L}(\mathsf{Head}, A, \mathtt{null})] * ((\mathtt{toadd}_\mathsf{t} = \mathsf{false}) \ltimes (\mathtt{toadd}_\mathsf{t} = \mathsf{true}))$

$\mathit{Add}_\mathsf{t} \stackrel{\text{def}}{=}$
$\exists x, y, z, n, v, w, u, tl, tl', n_1, S.$
$((\mathsf{L}_{\mathsf{t}::tl;n_1}(x, v, z) * (\mathtt{toadd}_\mathsf{t} = \mathsf{true}) * (\mathsf{S} = S)) \ltimes (\mathsf{L}_{\mathsf{t}::tl;n_1}(x, v, y) * \mathsf{U}(y, w, z) * (\mathtt{toadd}_\mathsf{t} = \mathsf{false}) * (\mathsf{S} = S \cup \{w\})))$
$* [\mathsf{N\_irr}_{\mathsf{t},tl'}(z, u, n) \wedge (v < w < u)]$

$\mathit{FailedAdd}_\mathsf{t} \stackrel{\text{def}}{=} (\mathtt{toadd}_\mathsf{t} = \mathsf{true}) \ltimes (\mathtt{toadd}_\mathsf{t} = \mathsf{false})$

$\mathit{Rmv}_\mathsf{t} \stackrel{\text{def}}{=}$
$\exists x, y, z, v, u, tl, tl', n_1, S.\ (\mathsf{L}_{\mathsf{t}::tl;n_1}(x, v, y) * \mathsf{L}_{\mathsf{t}::tl'}(y, u, z) * (\mathsf{S} = S \uplus \{u\}) \wedge (u < \mathtt{MAX})) \ltimes (\mathsf{L}_{\mathsf{t}::tl;n_1}(x, v, z) * (\mathsf{S} = S))$

$\mathit{ReqLockH}_\mathsf{t} \stackrel{\text{def}}{=}$
$\exists x, A, tl, L, n_1.\ (\mathsf{N\_irr}_{\mathsf{t},tl;n_1}(\mathsf{Head}, \mathtt{MIN}, x) \ltimes \mathsf{L}_{tl::\mathsf{t};n_1}(\mathsf{Head}, \mathtt{MIN}, x))$
$* [\mathsf{ls\_irr}_{\mathsf{t},L}(x, A, \mathtt{null})]$

$\mathit{ReqLock2}_\mathsf{t} \stackrel{\text{def}}{=}$
$\exists x, y, z, A, v, u, A', L, tl, tl', L', n_1.\ [\mathsf{ls\_irr}_{\mathsf{t},L}(\mathsf{Head}, A, x) * \mathsf{L}_{\mathsf{t}::tl}(x, v, y)]$
$* (\mathsf{N\_irr}_{\mathsf{t},tl';n_1}(y, u, z) \ltimes \mathsf{L}_{tl'::\mathsf{t};n_1}(y, u, z)) * [(u < \mathtt{MAX}) \wedge \mathsf{ls\_irr}_{\mathsf{t},L'}(z, A', \mathtt{null})]$

$\mathit{Unlock}_\mathsf{t} \stackrel{\text{def}}{=}$
$\exists x, y, A, v, L, tl, n_1.\ [\mathsf{ls\_irr}_{\mathsf{t},L}(\mathsf{Head}, A, x)] * (\mathsf{L}_{\mathsf{t}::tl;n_1}(x, v, y) \ltimes \mathsf{N\_irr}_{\mathsf{t},tl;n_1+1}(x, v, y))$

$\mathcal{D}_\mathsf{t} \stackrel{\text{def}}{=} (\forall x, n_1, v.\ \mathcal{B}_\mathsf{t}(x, n_1, v)) \wedge \mathcal{B}_0 \qquad \mathit{LayFeat} \stackrel{\text{def}}{=} \mathit{Int} \mid \bot \qquad \mathit{Layer} \stackrel{\text{def}}{=} \mathit{Int} \mid \mathtt{MIN}$

$\mathcal{B}_\mathsf{t}(x, n_1, v) \stackrel{\text{def}}{=} (dp_\mathsf{t}(x, n_1, v) \rightsquigarrow dq_\mathsf{t}(x, n_1, v), v) \qquad \mathcal{B}_0 \stackrel{\text{def}}{=} (\mathsf{False} \rightsquigarrow \mathsf{True}, \bot)$

$dp_\mathsf{t}(x, n_1, v) \stackrel{\text{def}}{=} \exists y, tl.\ \mathsf{L}_{\mathsf{t}::tl;n_1}(x, v, y) * \mathsf{true} \wedge I$

$dq_\mathsf{t}(x, n_1, v) \stackrel{\text{def}}{=} ((\exists y, tl.\ \mathsf{N\_irr}_{\mathsf{t},tl;n_1+1}(x, v, y)) \vee (\exists S.\ \mathsf{S} = S \wedge v \notin S)) * \mathsf{true} \wedge I$

$$\mathfrak{L}(\mathfrak{S}, \mathcal{B}) = \begin{cases} data & \text{if } \mathcal{B}.\mathcal{F} = data \\ \mathtt{MIN} & \text{if } \mathcal{B}.\mathcal{F} = \bot \\ undefined & \text{otherwise} \end{cases}$$

$layer1 > layer2$ iff
$(layer1 \in Int \wedge layer2 \in Int \wedge layer1 < layer2) \vee (layer1 \in Int \wedge layer2\ is\ \mathtt{MIN})$

Fig. 42. Rely, guarantee and layered definite actions of the lock-coupling list.

$\mathsf{tL}_{tl_1,\mathsf{t},tl_2;n_1,n}(p,v,y) \stackrel{\text{def}}{=} \mathsf{tlocked}_{tl_1,\mathsf{t},tl_2;n_1,n}(p) * (p.\mathtt{data} = v) * (p.\mathtt{next} = y)$

$\mathsf{tlocked}_{tl_1,\mathsf{t},tl_2;n_1,n}(p) \stackrel{\text{def}}{=}$
$\exists n_2.\ (\mathsf{t} \notin tl_1) \wedge (\mathsf{t} \notin tl_2) \wedge ((p.\mathtt{lowner} = n_1) * (p.\mathtt{lnext} = n_2) \wedge (n_1 \le n < n_2))$
$* \mathsf{tickets}(p, 0, n_1) * \mathsf{tickets}_{tl_1}(p, n_1, n) * \mathsf{tickets}_{\mathsf{t}::tl_2}(p, n, n_2) * \mathsf{tickets\_new}(p, n_2)$

$P_1' \stackrel{\text{def}}{=} \exists z.\ P_1'(z)$

$P_1'(z) \stackrel{\text{def}}{=}$
$\exists A, tl, L.\ \mathsf{toadd}_{\mathsf{cid}}(\mathsf{true}) * \mathsf{L}_{\mathsf{cid}::tl}(\mathsf{Head}, \mathtt{MIN}, z) * \mathsf{ls\_irr}_{\mathsf{cid},L}(z, A, \mathtt{null}) * \mathsf{ss}(\mathtt{MIN}::A)$
$\wedge (\mathtt{MIN} < \mathsf{e} < \mathtt{MAX}) \wedge (\mathsf{p} = \mathsf{Head})$

$P_3 \stackrel{\text{def}}{=}$
$\exists z, A_1, v, A_2, L_1, tl, tl', L_2.\ \mathsf{toadd}_{\mathsf{cid}}(\mathsf{true}) * \mathsf{ls\_irr}_{\mathsf{cid},L_1}(\mathsf{Head}, A_1, \mathsf{p}) * \mathsf{L}_{\mathsf{cid}::tl}(\mathsf{p}, v, \mathsf{c})$
$* \mathsf{N\_irr}_{\mathsf{cid},tl'}(\mathsf{c}, \mathsf{u}, z) * \mathsf{ls\_irr}_{\mathsf{cid},L_2}(z, A_2, \mathtt{null}) * \mathsf{ss}(A_1::v::\mathsf{u}::A_2) \wedge (v < \mathsf{e} < \mathtt{MAX})$
$\wedge (1 + \mathsf{len}(A_2) + |\mathsf{toaddThrds}(\mathsf{tidset}(tl' + L_2))| \le \mathsf{aux})$

$p_3'(c, z) \stackrel{\text{def}}{=}$
$\exists A_1, A_2, L_1, tl, L_2.\ \mathsf{toadd}_{\mathsf{cid}}(\mathsf{true}) * \mathsf{ls\_irr}_{\mathsf{cid},L_1}(\mathsf{Head}, A_1, c) * \mathsf{L}_{\mathsf{cid}::tl}(c, \mathsf{u}, z) * \mathsf{ss}(A_1::\mathsf{u}::A_2)$
$* \mathsf{ls\_irr}_{\mathsf{cid},L_2}(z, A_2, \mathtt{null}) \wedge\ (1 + \mathsf{len}(A_2) + |\mathsf{toaddThrds}(\mathsf{tidset}(L_2))| \le \mathsf{aux}) \wedge \Diamond(\mathsf{aux} - 1)$
$\wedge (\mathsf{u} < \mathsf{e} < \mathtt{MAX}) \wedge \mathsf{arem}(\mathtt{ADD}) \wedge (\mathsf{e} = \mathsf{E})$

```
add(e) {
 1  local p, c, x, pi, po, ci, co, u, r, aux;
    {P ∧ (MIN < e < MAX) ∧ arem(ADD) ∧ (e = E)}
 2  <p := Head;  toadd_cid := true>;
    {∃z, A, tl, L. toadd_cid(true) * N_irr_cid,tl(Head, MIN, z) * ls_irr_cid,L(z, A, null) * ss(MIN::A)
      ∧ (p = Head) ∧ (MIN < e < MAX) ∧ arem(ADD) ∧ (e = E)}
 3  lock(p);
    {P'_1 ∧ arem(ADD) ∧ (e = E)}
 4  c := p.next;
    {P'_1(c) ∧ arem(ADD) ∧ (e = E)}
 5  <u := c.data;  aux := metric(p)>;
    {P_3 ∧ arem(ADD) ∧ (e = E)}
    {P_3 ∧ arem(ADD) ∧ (e = E) ∧ ◇(aux)}
 6  while (u < e) {
      {P_3 ∧ ◇(aux − 1) ∧ (u < e) ∧ arem(ADD) ∧ (e = E)}
 7    lock(c);
      {∃z. p'_3(c, z)}
 8    unlock(p);
 9    p := c;
10    c := p.next;
      {p'_3(p, c)}
11    <u := c.data;  aux := metric(p)>;
      {P_3 ∧ arem(ADD) ∧ (e = E) ∧ ◇(aux)}
12  }
    {P_3 ∧ (e ≤ u) ∧ arem(ADD) ∧ (e = E)}
    {∃z, A_1, v, A_2, L_1, tl, tl', L_2. toadd_cid(true) * ls_irr_cid,L_1(Head, A_1, p) * L_cid::tl(p, v, c)
      * N_irr_cid,tl'(c, u, z) * ls_irr_cid,L_2(z, A_2, null) * ss(A_1::v::u::A_2) ∧ (v < e ≤ u)
      ∧(e < MAX) ∧ arem(ADD) ∧ (e = E)}
13  if (u != e) {
```

Fig. 43. Proof outline for add (1).

```
13  if (u != e) {
```
$\{\exists z, A_1, v, A_2, L_1, tl, tl', L_2.\ \mathsf{toadd_{cid}}(\mathsf{true}) * \mathsf{ls\_irr_{cid,L_1}}(\mathsf{Head}, A_1, \mathsf{p}) * \mathsf{L_{cid::tl}}(\mathsf{p}, v, \mathsf{c}) * \mathsf{N\_irr_{cid,tl'}}(\mathsf{c}, \mathsf{u}, z) * \mathsf{ls\_irr_{cid,L_2}}(z, A_2, \mathsf{null}) * \mathsf{ss}(A_1 :: v :: \mathsf{u} :: A_2) \wedge (v < \mathsf{e} < \mathsf{u}) \wedge \mathsf{arem}(\mathsf{ADD}) \wedge (\mathsf{e} = \mathsf{E})\}$
```
14    x := cons(0, 0, e, c);
```
$\{\exists z, A_1, v, A_2, L_1, tl, tl', L_2.\ \mathsf{toadd_{cid}}(\mathsf{true}) * \mathsf{ls\_irr_{cid,L_1}}(\mathsf{Head}, A_1, \mathsf{p}) * \mathsf{L_{cid::tl}}(\mathsf{p}, v, \mathsf{c}) * \mathsf{U}(\mathsf{x}, \mathsf{e}, \mathsf{c}) * \mathsf{N\_irr_{cid,tl'}}(\mathsf{c}, \mathsf{u}, z) * \mathsf{ls\_irr_{cid,L_2}}(z, A_2, \mathsf{null}) * \mathsf{ss}(A_1 :: v :: \mathsf{u} :: A_2) \wedge (v < \mathsf{e} < \mathsf{u}) \wedge \mathsf{arem}(\mathsf{ADD}) \wedge (\mathsf{e} = \mathsf{E})\}$
```
15    <p.next := x;  toadd_cid := false>;
```
$\{\exists z, A_1, v, A_2, L_1, tl, tl', tl'', L_2.\ \mathsf{toadd_{cid}}(\mathsf{false}) * \mathsf{ls\_irr_{cid,L_1}}(\mathsf{Head}, A_1, \mathsf{p}) * \mathsf{L_{cid::tl}}(\mathsf{p}, v, \mathsf{x}) * \mathsf{N\_irr_{cid,tl'}}(\mathsf{x}, \mathsf{e}, \mathsf{c}) * \mathsf{N\_irr_{cid,tl''}}(\mathsf{c}, \mathsf{u}, z) * \mathsf{ls\_irr_{cid,L_2}}(z, A_2, \mathsf{null}) * \mathsf{ss}(A_1 :: v :: \mathsf{e} :: \mathsf{u} :: A_2) \wedge \mathsf{arem}(\mathbf{skip}) \wedge (\mathsf{R} = \mathsf{true})\}$
```
16    r := true;
17  } else {
18    <r := false;  toadd_cid := false>;
19  }
```
$\{\exists z, A_1, v, A_2, L_1, tl, L_2.\ \mathsf{toadd_{cid}}(\mathsf{false}) * \mathsf{ls\_irr_{cid,L_1}}(\mathsf{Head}, A_1, \mathsf{p}) * \mathsf{L_{cid::tl}}(\mathsf{p}, v, z) * \mathsf{ls\_irr_{cid,L_2}}(z, A_2, \mathsf{null}) * \mathsf{ss}(A_1 :: v :: A_2) \wedge \mathsf{arem}(\mathbf{skip}) \wedge (\mathsf{r} = \mathsf{R})\}$
```
20  unlock(p);
```
$\{\exists A, L.\ \mathsf{toadd_{cid}}(\mathsf{false}) * \mathsf{ls\_irr_{cid,L}}(\mathsf{Head}, A, \mathsf{null}) * \mathsf{ss}(A) \wedge \mathsf{arem}(\mathbf{skip}) \wedge (\mathsf{r} = \mathsf{R})\}$

$\{P \wedge \mathsf{arem}(\mathbf{skip}) \wedge (\mathsf{r} = \mathsf{R})\}$
```
21  return r;
}
```

Fig. 44. Proof outline for add (2).

```
rmv(e) {
 1  local p, c, n, pi, po, ci, co, u, r, aux;
    {P ∧ (MIN < e < MAX) ∧ arem(RMV) ∧ (e = E)}
 2  p := Head;
 .  ...
    {∃z, A1, v, A2, L1, tl, tl', L2. toadd_cid(false) * ls_irr_cid,L1(Head, A1, p) * L_cid::tl(p, v, c)
      * N_irr_cid,tl'(c, u, z) * ls_irr_cid,L2(z, A2, null) * ss(A1 :: v :: u :: A2) ∧ (v < e ≤ u) ∧ (e < MAX)
      ∧arem(RMV) ∧ (e = E)}
16  if (u = e) {
      {∃z, A1, v, A2, L1, tl, tl', L2. toadd_cid(false) * ls_irr_cid,L1(Head, A1, p) * L_cid::tl(p, v, c)
       * N_irr_cid,tl'(c, e, z) * ls_irr_cid,L2(z, A2, null) * ss(A1 :: v :: e :: A2) ∧ (e < MAX) ∧ arem(RMV) ∧ (e = E)}
17    lock(c);
      {P'4 ∧ arem(RMV) ∧ (e = E)}
18    n := c.next;
      {∃A1, v, A2, L1, tl, L2. toadd_cid(false) * ls_irr_cid,L1(Head, A1, p) * L_cid::tl(p, v, c)
       * L_cid(c, e, n) * ls_irr_cid,L2(n, A2, null) * ss(A1 :: v :: e :: A2) ∧ (e < MAX) ∧ arem(RMV) ∧ (e = E)}
19    p.next := n;
      {∃A1, v, A2, L1, tl, L2. toadd_cid(false) * ls_irr_cid,L1(Head, A1, p) * L_cid::tl(p, v, n)
       * L_cid(c, e, n) * ls_irr_cid,L2(n, A2, null) * ss(A1 :: v :: A2) ∧ arem(skip) ∧ (R = true)}
20    unlock(p);
      {∃A, L. toadd_cid(false) * ls_irr_cid,L(Head, A, null) * L_cid(c, e, n) * ss(A) ∧ arem(skip) ∧ (R = true)}
21    dispose(c);
      {∃A, L. toadd_cid(false) * ls_irr_cid,L(Head, A, null) * ss(A) ∧ arem(skip) ∧ (R = true)}
22    r := true;
23  } else {
      {∃z, A1, v, A2, L1, tl, L2. toadd_cid(false) * ls_irr_cid,L1(Head, A1, p) * L_cid::tl(p, v, z)
       * ls_irr_cid,L2(z, A2, null) * ss(A1 :: v :: A2) ∧ arem(skip) ∧ (R = false)}
24    unlock(p);
      {∃A, L. toadd_cid(false) * ls_irr_cid,L(Head, A, null) * ss(A) ∧ arem(skip) ∧ (R = false)}
25    r := false;
26  }
    {∃A, L. toadd_cid(false) * ls_irr_cid,L(Head, A, null) * ss(A) ∧ arem(skip) ∧ (r = R)}
    {P ∧ arem(skip) ∧ (r = R)}
27  return r;
}
```

Fig. 45. Proof outline for rmv.

C.4 Optimistic list with TAS lock

Fig. 46 and 47 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. In Fig. 47, the thread locks only one node p before validation. It does not lock c until the thread is going to remove c at line 27. We prove it is deadlock-free.

Fig. 48 defines the precise invariant and the precondition. Fig. 49 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);
}
```

Fig. 46. Data structure for optimistic list with TAS lock.

As for the lock-coupling list, the invariant I defined in Figure 48 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 29 in Fig. 47). The precise invariant I should include those nodes (garb).

As shown in Figure 49, 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 describe the release of each lock. As usual, the progress proof of lock statement can be reused, and is similar to the TAS lock proof for nested locking object. The layers are the same as lock coupling list, and represents the reverse relation of *Int*.

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. 50 and Fig. 51 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 [Liang et al. 2014], 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. 52 is similar to the proof of the add method.

```
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    lock(p);
13    v := p.data;
14    s := Head;
15    w := s.data;
16    while (w < v) {
17      s := s.next;
18      w := s.data;
19    }
20    if (s = p && p.next = c) {
21      done := true;
22    } else {
23      unlock(p);
24    }
25  }
26  if (u != e) {
27    x := cons(0, e, c);
28    p.next := x;
29    r := true;
30  } else {
31    r := false;
32  }
33  unlock(p);
34  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    lock(p);
13    v := p.data;
14    s := Head;
15    w := s.data;
16    while (w < v) {
17      s := s.next;
18      w := s.data;
19    }
20    if (s = p && p.next = c) {
21      done := true;
22    } else {
23      unlock(p);
24    }
25  }
26  if (u = e) {
27    lock(c);
28    n := c.next;
29    <p.next := n; gn := gn ∪ {c}>;
30    unlock(c);
31    r := true;
32  } else {
33    r := false;
34  }
35  unlock(p);
36  return r;
}
```

Fig. 47. Add and rmv method of optimistic list with TAS lock.

$A ::= \epsilon \mid v :: A \qquad s ::= 0 \mid \mathsf{t} \qquad L ::= \epsilon \mid s :: L \qquad B \in \mathit{ThrdID} \rightharpoonup \mathit{Bool}$

$|\emptyset| \stackrel{\text{def}}{=} 0 \qquad |S \cup \{x\}| \stackrel{\text{def}}{=} |S| + 1 \qquad \mathsf{len}(\epsilon) \stackrel{\text{def}}{=} 0 \qquad \mathsf{len}(v :: A) \stackrel{\text{def}}{=} \mathsf{len}(A) + 1$

$I \stackrel{\text{def}}{=} \exists A, L.\ \mathsf{ls}_L(\mathsf{Head}, A, \mathtt{null}) * \mathsf{ss}(A) * \mathsf{garb}$

$P_{\mathsf{t}} \stackrel{\text{def}}{=} \exists A, L.\ \mathsf{ls\_irr}_{\mathsf{t},L}(\mathsf{Head}, A, \mathtt{null}) * \mathsf{ss}(A) * \mathsf{garb}$

$\mathsf{garb} \stackrel{\text{def}}{=} \circledast_{x \in \mathsf{gn}} \mathsf{N}\_(x, \_, \_)$

$\mathsf{ls}_L(x, A, y) \stackrel{\text{def}}{=}$
$\quad (x = y \wedge A = \epsilon \wedge L = \epsilon \wedge \mathsf{emp})$
$\quad \vee\ (x \neq y \wedge \exists z, v, A', s, L'.\ A = v :: A' \wedge L = s :: L' \wedge \mathsf{N}_s(x, v, z) * \mathsf{ls}_{L'}(z, A', y))$

$\mathsf{ls\_irr}_{\mathsf{t},L}(x, A, y) \stackrel{\text{def}}{=}$
$\quad (x = y \wedge A = \epsilon \wedge L = \epsilon \wedge \mathsf{emp})$
$\quad \vee\ (x \neq y \wedge \exists z, v, A', s, L'.\ A = v :: A' \wedge L = s :: L' \wedge \mathsf{N\_irr}_{\mathsf{t},s}(x, v, z) * \mathsf{ls\_irr}_{\mathsf{t},L'}(z, A', y))$

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

$\mathsf{N}_s(p, v, y) \stackrel{\text{def}}{=} \mathsf{lock}_s(p) * (p.\mathtt{data} = v) * (p.\mathtt{next} = y)$

$\mathsf{N\_irr}_{\mathsf{t},s}(p, v, y) \stackrel{\text{def}}{=} \mathsf{lock\_irr}_{\mathsf{t},s}(p) * (p.\mathtt{data} = v) * (p.\mathtt{next} = y)$

$\mathsf{L}_s(p, v, y) \stackrel{\text{def}}{=} \mathsf{locked}_s(p) * (p.\mathtt{data} = v) * (p.\mathtt{next} = y)$

$\mathsf{L\_irr}_{\mathsf{t},s}(p, v, y) \stackrel{\text{def}}{=} (s \neq \mathsf{t}) \wedge \mathsf{L}_s(p, v, y)$

$\mathsf{U}(p, v, y) \stackrel{\text{def}}{=} \mathsf{unlocked}(p) * (p.\mathtt{data} = v) * (p.\mathtt{next} = y)$

$\mathsf{lock}_s(p) \stackrel{\text{def}}{=} \mathsf{locked}_s(p) \vee (s = 0 \wedge \mathsf{unlocked}(p)) \qquad \mathsf{lock\_irr}_{\mathsf{t},s}(p) \stackrel{\text{def}}{=} \mathsf{lock}_s(p) \wedge (\mathsf{t} \neq s)$

$\mathsf{locked}_{\mathsf{t}}(p) \stackrel{\text{def}}{=} (p.\mathtt{lock} = \mathsf{t}) \wedge (\mathsf{t} \in \mathtt{TIDS}) \qquad \mathsf{unlocked}(p) \stackrel{\text{def}}{=} (p.\mathtt{lock} = 0)$

$\mathsf{s}(A) \stackrel{\text{def}}{=} \exists A'.\ (A = \mathtt{MIN} :: A' :: \mathtt{MAX}) \wedge \mathsf{sorted}(A)$

$\mathsf{ss}(A) \stackrel{\text{def}}{=} \exists A'.\ (A = \mathtt{MIN} :: A' :: \mathtt{MAX}) \wedge \mathsf{sorted}(A) \wedge (\mathtt{S} = A')$

$$\mathsf{sorted}(A) \stackrel{\text{def}}{=} \begin{cases} \mathsf{true} & \text{if } A = \epsilon \vee A = v :: \epsilon \\ (v_1 < v_2) \wedge \mathsf{sorted}(v_2 :: A') & \text{if } A = v_1 :: v_2 :: A' \end{cases}$$

Fig. 48. Invariant and precondition of the optimistic list with TAS lock.

$$R_t \stackrel{\text{def}}{=} \bigvee_{t' \neq t} G_{t'}$$

$$G_t \stackrel{\text{def}}{=} (Add_t \vee Rmv_t \vee Lock_t \vee Unlock_t \vee \mathsf{Id}) * \mathsf{Id} \wedge (I \ltimes I)$$

$$\begin{aligned} Add_t \stackrel{\text{def}}{=}\ & \exists x, y, z, n, v, w, u, s, S.\ ((\mathsf{L}_t(x, v, z) * (\mathsf{S} = S)) \ltimes_2 (\mathsf{L}_t(x, v, y) * \mathsf{U}(y, w, z) * (\mathsf{S} = S \cup \{w\}))) \\ & * [\mathsf{N\_irr}_{t,s}(z, u, n) \wedge (v < w < u)] \end{aligned}$$

$$\begin{aligned} Rmv_t \stackrel{\text{def}}{=}\ & \exists x, y, z, v, u, S, S_g.\ (\mathsf{L}_t(x, v, y) * (\mathsf{gn} = S_g) * (\mathsf{S} = S \uplus \{u\})) \\ & \ltimes_2 (\mathsf{L}_t(x, v, z) * (\mathsf{gn} = S_g \cup \{y\}) * (\mathsf{S} = S)) * [\mathsf{L}_t(y, u, z) \wedge (u < \mathtt{MAX})] \end{aligned}$$

$$Lock_t \stackrel{\text{def}}{=} \exists x, y, v.\ \mathsf{U}(x, v, y) \ltimes_1 \mathsf{L}_t(x, v, y)$$

$$Unlock_t \stackrel{\text{def}}{=} \exists x, y, v.\ \mathsf{L}_t(x, v, y) \ltimes \mathsf{U}(x, v, y)$$

$$\mathcal{D}_{\mathtt{cid}} \stackrel{\text{def}}{=} (\forall x, v.\ \mathcal{B}_{\mathtt{cid}}(x, v)) \wedge \mathcal{B}_0 \qquad LayFeat \stackrel{\text{def}}{=} Int \mid \bot \qquad Layer \stackrel{\text{def}}{=} Int \mid \mathtt{MIN}$$

$$\mathcal{B}_t(x, v) \stackrel{\text{def}}{=} (dp_t(x, v) \rightsquigarrow dq(x, v), v) \qquad \mathcal{B}_0 \stackrel{\text{def}}{=} (\mathsf{False} \rightsquigarrow \mathsf{True}, \bot)$$

$$dp_t(x, v) \stackrel{\text{def}}{=} \exists y.\ \mathsf{L}_t(x, v, y) * \mathsf{true} \wedge I \qquad dq(x) \stackrel{\text{def}}{=} \exists y.\ \mathsf{U}(x, v, y) * \mathsf{true} \wedge I$$

$$\mathfrak{L}(\mathfrak{S}, \mathcal{B}) = \begin{cases} data & \text{if } \mathcal{B}.\mathcal{F} = data \\ \mathtt{MIN} & \text{if } \mathcal{B}.\mathcal{F} = \bot \\ undefined & \text{otherwise} \end{cases}$$

$layer1 > layer2$ iff
$(layer1 \in Int \wedge layer2 \in Int \wedge layer1 < layer2) \vee (layer1 \in Int \wedge layer2\ is\ \mathtt{MIN})$

Fig. 49. Rely, guarantee and definite actions of the optimistic list with TAS lock.

$\mathsf{findLocked}(x, v, y, u) \stackrel{\text{def}}{=}$
$\exists z, A, A', s, L.\ \mathsf{ls\_irr}_{\mathsf{cid},L}(\mathsf{Head}, A, x) * \mathsf{L}_{\mathsf{cid}}(x, v, y) * \mathsf{N\_irr}_{\mathsf{cid},s}(y, u, z)$
$* \mathsf{ls\_irr}_{\mathsf{cid},L}(z, A', \mathtt{null}) * \mathsf{ss}(A :: v :: u :: A') * \mathsf{garb}$

$p_1(v, A_2, s_1) \stackrel{\text{def}}{=}$
$\exists z, A_1, L_1, s, L_2.\ \mathsf{ls\_irr}_{\mathsf{cid},L_1}(\mathsf{Head}, A_1, \mathsf{p}) * \mathsf{N}_{s_1}(\mathsf{p}, v, \mathsf{c}) * \mathsf{N\_irr}_{\mathsf{cid},s}(\mathsf{c}, \mathsf{u}, z) *$
$\mathsf{ls\_irr}_{\mathsf{cid},L_2}(z, A_2, \mathtt{null}) * \mathsf{ss}(A_1 :: v :: \mathsf{u} :: A_2) * \mathsf{garb}$

$p_1'(v, A_2, s_1) \stackrel{\text{def}}{=}$
$\exists x, z, s, L_2.\ \mathsf{N}_{s_1}(\mathsf{p}, v, x) * \mathsf{N\_irr}_{\mathsf{cid},s}(\mathsf{c}, \mathsf{u}, z) * \mathsf{ls\_irr}_{\mathsf{cid},L_2}(z, A_2, \mathtt{null}) * \mathsf{true}$
$\wedge\ (\mathsf{p} \in \mathsf{gn} \vee \mathsf{c} \in \mathsf{gn} \vee \exists v', A', L'.\ \mathsf{ls\_irr}_{\mathsf{cid},L'}(x, v' :: A', \mathsf{c}) * \mathsf{true})$

$p_2(v) \stackrel{\text{def}}{=} \exists A_2.\ p_2(v, A_2)$ $\qquad$ $p_2(v, A_2) \stackrel{\text{def}}{=} \exists s_1.\ P \wedge p_1(v, A_2, s_1) \wedge (s_1 \neq \mathsf{cid})$

$p_3(v) \stackrel{\text{def}}{=} \exists A_2.\ P' \wedge p_1(v, A_2, \mathsf{cid})$

$p_2'(v) \stackrel{\text{def}}{=} \exists A_2.\ p_2'(v, A_2)$ $\qquad$ $p_2'(v, A_2) \stackrel{\text{def}}{=} \exists s_1.\ P \wedge p_1'(v, A_2, s_1) \wedge (s_1 \neq \mathsf{cid})$

```
add(e) {
 1  local p, c, x, s, done, b, w, v, u, r;
    {P ∧ ♦(1,1) ∧ arem(ADD) ∧ (MIN < e < MAX) ∧ (e = E)}
 2  done := false;
    {(¬done ∧ P ∧ ♦(1,1) ∧ ◇(1) ∨ ∃v. done ∧ findLocked(p, v, c, u)
     ∧♦(1,0) ∧ (v < e ≤ u)) ∧ arem(ADD) ∧ (MIN < e < MAX) ∧ (e = E)}
 3  while (!done) {
      {¬done ∧ P ∧ ♦(1,1) ∧ arem(ADD) ∧ (MIN < e < MAX) ∧ (e = E)}
 4    p := Head;
 5    c := p.next;
 6    u := c.data;
      {¬done ∧ ∃v, A2. (p2(v, A2) ∧ ♦(1,1) ∨ p2'(v, A2) ∧ ♦(1,2) ∧ ◇(1))
       ∧(v < e < MAX) ∧ arem(ADD) ∧ (e = E)}
      {¬done ∧ ∃v, A2. (p2(v, A2) ∧ ♦(1,1) ∨ p2'(v, A2) ∧ ♦(1,2)) ∧ ◇(1 + len(A2))
       ∧(v < e < MAX) ∧ arem(ADD) ∧ (e = E)}
 7    while (u < e) {
        {¬done ∧ ∃v, A2. (p2(v, A2) ∧ ♦(1,1) ∨ p2'(v, A2) ∧ ♦(1,2)) ∧ ◇(len(A2))
         ∧(u < e < MAX) ∧ arem(ADD) ∧ (e = E)}
 8      p := c;
 9      c := c.next;
10      u := c.data;
        {¬done ∧ ∃v, A2. (p2(v, A2) ∧ ♦(1,1) ∨ p2'(v, A2) ∧ ♦(1,2)) ∧ ◇(1 + len(A2))
         ∧(v < e < MAX) ∧ arem(ADD) ∧ (e = E)}
11    }
    {¬done ∧ ∃v. (p2(v) ∧ ♦(1,1) ∨ p2'(v) ∧ ♦(1,2) ∧ ◇(1)) ∧ (v < e ≤ u) ∧ arem(ADD) ∧ (e = E)}
```

Fig. 50. Proof outline of add (1).

$p_3'(v) \stackrel{\text{def}}{=} \exists A_2.\ P' \wedge p_1'(v, A_2, \mathsf{cid})$

$p_4(A_2) \stackrel{\text{def}}{=}$
$\exists z, A_1, L_1, s, L_2.\ \mathsf{ls}_{L_1}(\mathsf{Head}, A_1, \mathsf{s}) * \mathsf{N}_s(\mathsf{s}, \mathsf{w}, z) * \mathsf{ls}_{L_2}(z, A_2, \mathtt{null}) * \mathsf{ss}(A_1 :: \mathsf{w} :: A_2) * \mathsf{garb}$

$p_4'(A_2) \stackrel{\text{def}}{=}$
$\exists z, s, L_2.\ \mathsf{N\_irr}_{\mathsf{cid},s}(\mathsf{s}, \mathsf{w}, z) * \mathsf{ls\_irr}_{\mathsf{cid},L_2}(z, A_2, \mathtt{null}) * \mathsf{true} \wedge (\mathsf{s} \in \mathsf{gn})$

$p_5 \stackrel{\text{def}}{=} \exists v, x, e.\ \mathsf{findLocked}(\mathsf{p}, v, x, e)$

```
12    lock(p);
      {¬done ∧ ∃v. (p3(v) ∧ ♦(1,0) ∨ p3'(v) ∧ ♦(1,1) ∧ ◇(1)) ∧ (v < e ≤ u) ∧ arem(ADD) ∧ (e = E)}
13    v := p.data;
14    s := Head;
15    w := s.data;
      {¬done ∧ ∃A2. (p4(A2) ∨ p4'(A2)) ∧ (p3(v) ∧ ♦(1,0) ∨ p3'(v) ∧ ♦(1,1) ∧ ◇(1))
       ∧(v < e ≤ u) ∧ arem(ADD) ∧ (e = E)}
      {¬done ∧ ∃A2. (p4(A2) ∨ p4'(A2)) ∧ ◇(1 + len(A2)) ∧ (p3(v) ∧ ♦(1,0) ∨ p3'(v) ∧ ♦(1,1))
       ∧(v < e ≤ u) ∧ arem(ADD) ∧ (e = E)}
16    while (w < v) {
17      s := s.next;
18      w := s.data;
19    }
      {¬done ∧ ∃A2. (p4(A2) ∨ p4'(A2)) ∧ (p3(v) ∧ ♦(1,0) ∨ p3'(v) ∧ ♦(1,1) ∧ ◇(1))
       ∧(v ≤ w) ∧ (v < e ≤ u) ∧ arem(ADD) ∧ (e = E)}
20    if (s = p && p.next = c) {
        {¬done ∧ p3(v) ∧ ♦(1,0) ∧ (v < e ≤ u) ∧ arem(ADD) ∧ (e = E)}
21      done := true;
        {done ∧ findLocked(p, v, c, u) ∧ ♦(1,0)) ∧ (v < e ≤ u) ∧ arem(ADD) ∧ (e = E)}
22    } else {
23      unlock(p);
        {¬done ∧ P ∧ ♦(1,1) ∧ ◇(1) ∧ arem(ADD) ∧ (MIN < e < MAX) ∧ (e = E)}
24    }
25  }
    {∃v. findLocked(p, v, c, u) ∧ ♦(1,0) ∧ (v < e ≤ u) ∧ arem(ADD) ∧ (e = E)}
26  if (u != e) {
      {∃v. findLocked(p, v, c, u) ∧ ♦(1,0) ∧ (v < e < u) ∧ arem(ADD) ∧ (e = E)}
27    x := cons(0, e, c);
28    p.next := x;
29    r := true;
      {p5 ∧ arem(skip) ∧ (r = R)}
30  } else {
      {∃v. findLocked(p, v, c, u) ∧ ♦(1,0) ∧ (e = u) ∧ arem(ADD) ∧ (e = E)}
31    r := false;
      {p5 ∧ arem(skip) ∧ (r = R)}
32  }
    {p5 ∧ arem(skip) ∧ (r = R)}
33  unlock(p);
    {P ∧ arem(skip) ∧ (r = R)}
34  return r;
}
```

Fig. 51. Proof outline of add (2).

$$\mathsf{adjLocked}(x,v,y,u,z) \stackrel{\text{def}}{=} \exists A, A', L.\ \mathsf{ls\_irr}_{\mathsf{cid},L}(\mathsf{Head}, A, x) * \mathsf{L}_{\mathsf{cid}}(x,v,y) * \mathsf{L}_{\mathsf{cid}}(y,u,z) * \mathsf{ls\_irr}_{\mathsf{cid},L}(z, A', \mathtt{null}) * \mathsf{ss}(A :: v :: u :: A') * \mathsf{garb}$$

```
rmv(e) {
 1  local p, c, n, s, done, b, v, u, r;
    {P ∧ ♦(1,2) ∧ arem(RMV) ∧ (MIN < e < MAX) ∧ (e = E)}
 2  done := false;
 3  while (!done) {
..     ...
25  }
    {∃v. findLocked(p, v, c, u) ∧ ♦(1,1) ∧ (v < e ≤ u < MAX) ∧ arem(RMV) ∧ (e = E)}
26  if (u = e) {
      {findLocked(p, _, c, e) ∧ ♦(1,1) ∧ arem(RMV) ∧ (e = E < MAX)}
27    lock(c);

      {∃z. adjLocked(p, _, c, e, z) ∧ ♦(1,0) ∧ arem(RMV) ∧ (e = E < MAX)}
28    n := c.next;
      {adjLocked(p, _, c, e, n) ∧ ♦(1,0) ∧ arem(RMV) ∧ (e = E < MAX)}
29    <p.next := n;  gn := gn ∪ {c}>;
      {p5 ∧ arem(skip) ∧ (R = true)}
30    unlock(c);
31    r := true;
      {p5 ∧ arem(skip) ∧ (r = R)}
32  } else {
      {∃v. findLocked(p, v, c, u) ∧ ♦(1,1) ∧ (v < e < u) ∧ arem(RMV) ∧ (e = E)}
33    r := false;
      {p5 ∧ arem(skip) ∧ (r = R)}
34  }
    {p5 ∧ arem(skip) ∧ (r = R)}
35  unlock(p);
    {P ∧ arem(skip) ∧ (r = R)}
36  return r;
}
```

Fig. 52. Proof outline of rmv.

C.5 Lazy list with TAS lock

Fig. 53 and 54 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., [Liang and Feng 2013; Vafeiadis 2008]), and we omit its proofs here.

Fig. 55 defines the precise invariant and the precondition. Fig. 56 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);
}
```

Fig. 53. Data structure for lazy list with TAS lock.

As shown in Figure 56, 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*. 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 [Liang et al. 2014], we also provide the (FR-CONJ) rule to add back the original tokens of the outer loop.

```
add(e) {
 1  local p, c, x, done, b, 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    b := false;
17    while (!b) {
18      b := cas(c.lock, 0, cid);
19    }
20    if (!p.mark && !c.mark
              && p.next = c)
21      done := true;
22    else {
23      p.lock := 0;
24      c.lock := 0;
25    }
26  }
27  c.lock := 0;
28  if (u != e) {
29    x := cons(0, e, c, false);
30    p.next := x;
31    r := true;
32  } else {
33    r := false;
34  }
35  p.lock := 0;
36  return r;
}
```

```
rmv(e) {
 1  local p, c, n, done, b, 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    b := false;
17    while (!b) {
18      b := cas(c.lock, 0, cid);
19    }
20    if (!p.mark && !c.mark
              && p.next = c)
21      done := true;
22    else {
23      p.lock := 0;
24      c.lock := 0;
25    }
26  }
27  if (u = e) {
28    <c.mark := true;
      tmark_cid := true>;
29    n := c.next;
30    <p.next := n; gn := gn ∪ {c};
      tmark_cid := false>;
31    r := true;
32  } else {
33    r := false;
34  }
35  p.lock := 0;
36  c.lock := 0;
37  return r;
}
```

Fig. 54. Lazy list with TAS lock.

The rmv method is given $\blacklozenge(2, 2)$ initially, where the two 2-level $\blacklozenge$-token are for doing *Mark* and *Rmv*. The proof of rmv is similar to the proof of the add method.

$A ::= \epsilon \mid v :: A \qquad s ::= 0 \mid \mathsf{t} \qquad L ::= \epsilon \mid s :: L \qquad M ::= \epsilon \mid b :: M \qquad B \in \mathit{ThrdID} \rightharpoonup \mathit{Bool}$

$|\emptyset| \stackrel{\text{def}}{=} 0 \qquad |S \cup \{x\}| \stackrel{\text{def}}{=} |S| + 1 \qquad \mathsf{len}(\epsilon) \stackrel{\text{def}}{=} 0 \qquad \mathsf{len}(v :: A) \stackrel{\text{def}}{=} \mathsf{len}(A) + 1$

$I \stackrel{\text{def}}{=} \exists A, L.\ \mathsf{tmarks} * \mathsf{ls}_L(\mathsf{Head}, A, \mathtt{null}) * \mathsf{ss}(A) * \mathsf{garb}$

$P_{\mathsf{t}} \stackrel{\text{def}}{=} P_{\mathsf{t}}(\mathsf{false}) \qquad P_{\mathsf{t}}(b) \stackrel{\text{def}}{=} \exists A, L.\ \mathsf{tmarks}_{\mathsf{t}}(b) * \mathsf{ls\_irr}_{\mathsf{t},L}(\mathsf{Head}, A, \mathtt{null}) * \mathsf{ss}(A) * \mathsf{garb}$

$\mathsf{tmarks} \stackrel{\text{def}}{=} \exists B.\ \mathsf{tmarks}(B) \qquad \mathsf{tmarks}(B) \stackrel{\text{def}}{=} (\circledast_{\mathsf{t}} \mathtt{tmark}_{\mathsf{t}} = B(\mathsf{t}))$

$\mathsf{tmarks}_{\mathsf{t}}(b) \stackrel{\text{def}}{=} \exists B.\ \mathtt{tmark}_{\mathsf{t}}(b, B) \qquad \mathsf{tmarks}_{\mathsf{t}}(b, B) \stackrel{\text{def}}{=} (\mathtt{tmark}_{\mathsf{t}} = b) * (\circledast_{\mathsf{t}' \neq \mathsf{t}} \mathtt{tmark}_{\mathsf{t}'} = B(\mathsf{t}'))$

$\mathsf{garb} \stackrel{\text{def}}{=} \circledast_{x \in \mathsf{gn}} \mathsf{N\_}(x, \_, \_, \_)$

$\mathsf{ls}_L(x, A, y) \stackrel{\text{def}}{=} \exists M.\ \mathsf{ls}_L(x, A, y, M) \qquad \mathsf{ls\_irr}_{\mathsf{t},L}(x, A, y) \stackrel{\text{def}}{=} \exists M.\ \mathsf{ls\_irr}_{\mathsf{t},L}(x, A, y, M)$

$\mathsf{ls}_L(x, A, y, M) \stackrel{\text{def}}{=}$
$\quad (x = y \wedge A = \epsilon \wedge L = \epsilon \wedge M = \epsilon \wedge \mathsf{emp})$
$\quad \vee\ (x \neq y \wedge \exists z, v, A', s, L', b, M'.\ A = (v, b) :: A' \wedge L = s :: L' \wedge M = b :: M' \wedge \mathsf{N}_s(x, v, z, b) * \mathsf{ls}_{L'}(z, A', y, M'))$

$\mathsf{ls\_irr}_{\mathsf{t},L}(x, A, y, M) \stackrel{\text{def}}{=}$
$\quad (x = y \wedge A = \epsilon \wedge L = \epsilon \wedge M = \epsilon \wedge \mathsf{emp})$
$\quad \vee\ (x \neq y \wedge \exists z, v, A', s, L', b, M'.\ A = (v, b) :: A' \wedge L = s :: L' \wedge M = b :: M'$
$\quad \wedge \mathsf{N\_irr}_{\mathsf{t},s}(x, v, z, b) * \mathsf{ls\_irr}_{\mathsf{t},L'}(z, A', y, M'))$

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

$\mathsf{N}_s(p, v, y, b) \stackrel{\text{def}}{=} \mathsf{lock}_s(p) * (p.\mathtt{data} = v) * (p.\mathtt{next} = y) * (p.\mathtt{mark} = b)$

$\mathsf{N\_irr}_{\mathsf{t},s}(p, v, y, b) \stackrel{\text{def}}{=} \mathsf{lock\_irr}_{\mathsf{t},s}(p) * (p.\mathtt{data} = v) * (p.\mathtt{next} = y) * (p.\mathtt{mark} = b)$

$\mathsf{L}_s(p, v, y, b) \stackrel{\text{def}}{=} \mathsf{locked}_s(p) * (p.\mathtt{data} = v) * (p.\mathtt{next} = y) * (p.\mathtt{mark} = b)$

$\mathsf{L\_irr}_{\mathsf{t},s}(p, v, y, b) \stackrel{\text{def}}{=} (s \neq \mathsf{t}) \wedge \mathsf{L}_s(p, v, y, b)$

$\mathsf{U}(p, v, y, b) \stackrel{\text{def}}{=} \mathsf{unlocked}(p) * (p.\mathtt{data} = v) * (p.\mathtt{next} = y) * (p.\mathtt{mark} = b)$

$\mathsf{lock}_s(p) \stackrel{\text{def}}{=} \mathsf{locked}_s(p) \vee (s = 0 \wedge \mathsf{unlocked}(p)) \qquad \mathsf{lock\_irr}_{\mathsf{t},s}(p) \stackrel{\text{def}}{=} \mathsf{lock}_s(p) \wedge (\mathsf{t} \neq s)$

$\mathsf{locked}_{\mathsf{t}}(p) \stackrel{\text{def}}{=} (p.\mathtt{lock} = \mathsf{t}) \wedge (\mathsf{t} \in \mathtt{TIDS}) \qquad \mathsf{unlocked}(p) \stackrel{\text{def}}{=} (p.\mathtt{lock} = 0)$

$\mathsf{s}(A) \stackrel{\text{def}}{=} \exists A'.\ (A = (\mathtt{MIN}, \mathsf{false}) :: A' :: (\mathtt{MAX}, \mathsf{false})) \wedge \mathsf{sorted}(A)$

$\mathsf{ss}(A) \stackrel{\text{def}}{=} \exists A'.\ (A = (\mathtt{MIN}, \mathsf{false}) :: A' :: (\mathtt{MAX}, \mathsf{false})) \wedge \mathsf{sorted}(A) \wedge (\mathsf{S} = \mathsf{elems}(A'))$

$$\mathsf{sorted}(A) \stackrel{\text{def}}{=} \begin{cases} \mathsf{true} & \text{if } A = \epsilon \vee A = (v, b) :: \epsilon \\ (v_1 < v_2) \wedge \mathsf{sorted}((v_2, b_2) :: A') & \text{if } A = (v_1, b_1) :: (v_2, b_2) :: A' \end{cases}$$

$$\mathsf{elems}(A) \stackrel{\text{def}}{=} \begin{cases} \emptyset & \text{if } A = \epsilon \\ \{v\} \cup \mathsf{elems}(A') & \text{if } A = (v, \mathsf{true}) :: A' \\ \mathsf{elems}(A') & \text{if } A = (v, \mathsf{false}) :: A' \end{cases}$$

Fig. 55. Invariant and precondition of the lazy list with TAS lock.

$R_\mathsf{t} \stackrel{\text{def}}{=} \bigvee_{\mathsf{t}' \neq \mathsf{t}} G_{\mathsf{t}'}$

$G_\mathsf{t} \stackrel{\text{def}}{=} (\mathit{Add}_\mathsf{t} \vee \mathit{Mark}_\mathsf{t} \vee \mathit{Rmv}_\mathsf{t} \vee \mathit{Lock11}_\mathsf{t} \vee \mathit{Lock21}_\mathsf{t} \vee \mathit{Lock1}_\mathsf{t} \vee \mathit{Lock2}_\mathsf{t} \vee \mathit{Unlock}_\mathsf{t} \vee \mathsf{Id}) * \mathsf{Id} \wedge (I \ltimes I)$

$\mathit{Add}_\mathsf{t} \stackrel{\text{def}}{=}$
$\exists x, y, z, n, v, w, u, s, S.\ ((\mathsf{L}_\mathsf{t}(x, v, z, \mathsf{false}) * (\mathsf{S} = S)) \ltimes_2 (\mathsf{L}_\mathsf{t}(x, v, y, \mathsf{false}) * \mathsf{U}(y, w, z, \mathsf{false}) * (\mathsf{S} = S \cup \{w\})))$
$* [\mathsf{N\_irr}_{\mathsf{t},s}(z, u, n, \mathsf{false}) \wedge (v < w < u)]$

$\mathit{Mark}_\mathsf{t} \stackrel{\text{def}}{=}$
$\exists x, y, z, v, u, S.\ [\mathsf{L}_\mathsf{t}(x, v, y, \mathsf{false})] * ((\mathsf{L}_\mathsf{t}(y, u, z, \mathsf{false}) * (\mathtt{tmark}_\mathsf{t} = \mathsf{false}) * (\mathsf{S} = S \uplus \{u\}) \wedge (u < \mathtt{MAX}))$
$\ltimes_2 (\mathsf{L}_\mathsf{t}(y, u, z, \mathsf{true}) * (\mathtt{tmark}_\mathsf{t} = \mathsf{true}) * (\mathsf{S} = S)))$

$\mathit{Rmv}_\mathsf{t} \stackrel{\text{def}}{=}$
$\exists x, y, z, v, u, S_g.\ (\mathsf{L}_\mathsf{t}(x, v, y, \mathsf{false}) * (\mathtt{tmark}_\mathsf{t} = \mathsf{true}) * (\mathsf{gn} = S_g))$
$\ltimes_2 (\mathsf{L}_\mathsf{t}(x, v, z, \mathsf{false}) * (\mathtt{tmark}_\mathsf{t} = \mathsf{false}) * (\mathsf{gn} = S_g \cup \{y\})) * [\mathsf{L}_\mathsf{t}(y, u, z, \mathsf{true}) \wedge (u < \mathtt{MAX})]$

$\mathit{Lock11}_\mathsf{t} \stackrel{\text{def}}{=}$
$\exists x, y, A_1, v, A_2.\ [\mathsf{ls\_irr}_{\mathsf{t},L_1}(\mathsf{Head}, A_1, x) \wedge (v < \mathtt{MAX})] * (\mathsf{U}(x, v, y, \mathsf{false}) \ltimes_1 \mathsf{L}_\mathsf{t}(x, v, y, \mathsf{false}))$
$* [\mathsf{ls\_unlocked\_unmarked}(y, A_2, \mathtt{null})]$

$\mathit{Lock1}_\mathsf{t} \stackrel{\text{def}}{=}$
$\exists x, y, z, z', v, A_2, A_2', b, b', L_2, s, L_2'.\ (\mathsf{U}(x, v, y, b) \ltimes \mathsf{L}_\mathsf{t}(x, v, y, b))$
$* [\mathsf{ls\_irr}_{\mathsf{t},L_2}(y, A_2, z) * \mathsf{N\_irr}_{\mathsf{t},s}(z, \_, z', b') * \mathsf{ls\_irr}_{\mathsf{t},L_2'}(z', A_2', \mathtt{null}) \wedge (b = \mathsf{true} \vee b' = \mathsf{true} \vee s \neq 0)]$

$\mathit{Lock21}_\mathsf{t} \stackrel{\text{def}}{=}$
$\exists x, y, z, A_1, v, u, A_2, L_1, L_2, M_2.\ [\mathsf{ls\_irr}_{\mathsf{t},L_1}(\mathsf{Head}, A_1, x) * \mathsf{L}_\mathsf{t}(x, v, y, \mathsf{false}) \wedge (v < \mathtt{MAX})] *$
$(\mathsf{U}(y, u, z, \mathsf{false}) \ltimes_1 \mathsf{L}_\mathsf{t}(y, u, z, \mathsf{false})) * [\mathsf{ls\_irr}_{\mathsf{t},L_2}(z, A_2, \mathtt{null}, M_2) \wedge (\forall b \in M_2.\ b = \mathsf{false})]$

$\mathit{Lock2}_\mathsf{t} \stackrel{\text{def}}{=}$
$\exists x, y, y', z, v, u, A_2, b_1, b_2, L_2, M_2.\ [\mathsf{L}_\mathsf{t}(x, v, y, b_1) \wedge (v < \mathtt{MAX})] * (\mathsf{U}(y', u, z, b_2) \ltimes \mathsf{L}_\mathsf{t}(y', u, z, b_2))$
$* [\mathsf{ls\_irr}_{\mathsf{t},L_2}(z, A_2, \mathtt{null}, M_2) * \mathsf{true} \wedge (b_1 = \mathsf{true} \vee b_2 = \mathsf{true} \vee \mathsf{true} \in M_2$
$\vee \exists v', A', L'.\ \mathsf{ls\_irr}_{\mathtt{cid},L'}(y, (v', \_) :: A', y') * \mathsf{true})]$

$\mathit{Unlock}_\mathsf{t} \stackrel{\text{def}}{=}$
$\exists x, y, v, b.\ [\mathsf{tmarks}_\mathsf{t}(\mathsf{false})] * (\mathsf{L}_\mathsf{t}(x, v, y, b) \ltimes \mathsf{U}(x, v, y, b))$

$\mathcal{D}_\mathtt{cid} \stackrel{\text{def}}{=} (\forall x, v.\ \mathcal{B}_\mathtt{cid}(x, v)) \wedge \mathcal{B}_0 \qquad \mathit{LayFeat} \stackrel{\text{def}}{=} \mathit{Int} \mid \bot \qquad \mathit{Layer} \stackrel{\text{def}}{=} \mathit{Int} \mid \mathtt{MIN}$

$\mathcal{B}_\mathsf{t}(x, v) \stackrel{\text{def}}{=} (dp_\mathsf{t}(x, v) \rightsquigarrow dq(x, v), v) \qquad \mathcal{B}_0 \stackrel{\text{def}}{=} (\mathsf{False} \rightsquigarrow \mathsf{True}, \bot)$

$dp_\mathsf{t}(x, v) \stackrel{\text{def}}{=} \exists y.\ \mathsf{L}_\mathsf{t}(x, v, y) * \mathsf{true} \wedge I \qquad dq(x, v) \stackrel{\text{def}}{=} \exists y.\ \mathsf{U}(x, v, y) * \mathsf{true} \wedge I$

$$\mathfrak{L}(\mathfrak{S}, \mathcal{B}) = \begin{cases} \mathit{data} & \text{if } \mathcal{B}.\mathcal{F} = \mathit{data} \\ \mathtt{MIN} & \text{if } \mathcal{B}.\mathcal{F} = \bot \\ \mathit{undefined} & \text{otherwise} \end{cases}$$

$\mathit{layer1} > \mathit{layer2}$ iff
$(\mathit{layer1} \in \mathit{Int} \wedge \mathit{layer2} \in \mathit{Int} \wedge \mathit{layer1} < \mathit{layer2}) \vee (\mathit{layer1} \in \mathit{Int} \wedge \mathit{layer2}\ \mathit{is}\ \mathtt{MIN})$

Fig. 56. Rely, guarantee and definite actions of the lazy list with TAS lock.

$\mathsf{adjLocked}(x, v, y, u, b, b', b_m) \stackrel{\text{def}}{=}$
$\exists z, A, A', L.\ \mathsf{tmarks}(b_m) * \mathsf{ls\_irr}_{\mathtt{cid},L}(\mathsf{Head}, A, x) * \mathsf{L}_{\mathtt{cid}}(x, v, y, b) * \mathsf{L}_{\mathtt{cid}}(y, u, z, b')$
$* \mathsf{ls\_irr}_{\mathtt{cid},L}(z, A', \mathtt{null}) * \mathsf{ss}(A :: (v, b) :: (u, b') :: A') * \mathsf{garb}$

$p_0(b_m) \stackrel{\text{def}}{=} \exists A, L, M.\ \mathsf{tmarks}_{\mathsf{t}}(b_m) * \mathsf{ls\_irr}_{\mathsf{t},L}(\mathsf{Head}, A, \mathtt{null}, M) * \mathsf{ss}(A) * \mathsf{garb} \wedge (\forall b \in M_2.\ b = \mathsf{false})$

$p'_0(b_m) \stackrel{\text{def}}{=} \exists A, L, M.\ \mathsf{tmarks}_{\mathsf{t}}(b_m) * \mathsf{ls\_irr}_{\mathsf{t},L}(\mathsf{Head}, A, \mathtt{null}, M) * \mathsf{ss}(A) * \mathsf{garb} \wedge (\mathsf{true} \in M_2)$

$p_1(v, A_2, s_1, s_2) \stackrel{\text{def}}{=}$
$\exists z, A_1, L_1, L_2, M_2.\ \mathsf{tmarks}_{\mathsf{t}}(\mathsf{false}) * \mathsf{ls\_irr}_{\mathtt{cid},L_1}(\mathsf{Head}, A_1, \mathsf{p}) *$
$\mathsf{N}_{s_1}(\mathsf{p}, v, \mathsf{c}, \mathsf{false}) * \mathsf{N}_{s_2}(\mathsf{c}, \mathsf{u}, z, \mathsf{false}) * \mathsf{ls\_irr}_{\mathtt{cid},L_2}(z, A_2, \mathtt{null}, M_2) * \mathsf{true} \wedge\ (\forall b \in M_2.\ b = \mathsf{false})$

$p'_1(v, A_2, s_1, s_2) \stackrel{\text{def}}{=}$
$\exists x, z, L_2, b_1, b_2, M_2.\ \mathsf{tmarks}_{\mathsf{t}}(\mathsf{false}) * \mathsf{N}_{s_1}(\mathsf{p}, v, x, b_1) * \mathsf{N}_{s_2}(\mathsf{c}, \mathsf{u}, z, b_2) * \mathsf{ls\_irr}_{\mathtt{cid},L_2}(z, A_2, \mathtt{null}, M_2) * \mathsf{true}$
$\wedge\ (b_1 = \mathsf{true} \vee b_2 = \mathsf{true} \vee \mathsf{true} \in M_2 \vee \exists v', A', L'.\ \mathsf{ls\_irr}_{\mathtt{cid},L'}(x, (v', \_) :: A', \mathsf{c}) * \mathsf{true})$

$p_2(v) \stackrel{\text{def}}{=} \exists A_2.\ p_2(v, A_2)$ $\quad$ $p_2(v, A_2) \stackrel{\text{def}}{=} \exists s_1, s_2.\ P \wedge p_1(v, A_2, s_1, s_2) \wedge (s_1 \neq \mathtt{cid}) \wedge (s_2 \neq \mathtt{cid})$

$p'_2(v) \stackrel{\text{def}}{=} \exists A_2.\ p'_2(v, A_2)$ $\quad$ $p'_2(v, A_2) \stackrel{\text{def}}{=} \exists s_1, s_2.\ P \wedge p'_1(v, A_2, s_1, s_2) \wedge (s_1 \neq \mathtt{cid}) \wedge (s_2 \neq \mathtt{cid})$

$p_3(v) \stackrel{\text{def}}{=} \exists A_2, s_2.\ P' \wedge p_1(v, A_2, \mathtt{cid}, s_2) \wedge (s_2 \neq \mathtt{cid})$ $\quad$ $p'_3(v) \stackrel{\text{def}}{=} \exists A_2, s_2.\ P' \wedge p'_1(v, A_2, \mathtt{cid}, s_2) \wedge (s_2 \neq \mathtt{cid})$

$p_4(v) \stackrel{\text{def}}{=} \exists A_2.\ P'' \wedge p_1(\mathtt{cid}, \mathtt{cid})$ $\quad$ $p'_4(v) \stackrel{\text{def}}{=} \exists A_2.\ P'' \wedge p'_1(\mathtt{cid}, \mathtt{cid})$

```
add(e) {
 1  local p, c, x, done, b, u, r;
    {P ∧ ♦(1,2) ∧ arem(ADD) ∧ (MIN < e < MAX) ∧ (e = E)}
 2  done := false;
    {(¬done ∧ P ∧ ♦(1,2) ∧ ◇(1)  ∨  ∃v. done ∧ adjLocked(p, v, c, u, false, false, false)
    ∧♦(1,0) ∧ v < e ≤ u) ∧  arem(ADD) ∧ (MIN < e < MAX) ∧ (e = E)}
 3  while (!done) {
      {¬done ∧ (p0(false) ∨ p'0(false) ∧ ◇(1)) ∧ ♦(1,2) ∧ arem(ADD) ∧ (MIN < e < MAX) ∧ (e = E)}
 4    p := Head;
 5    c := p.next;
 6    u := c.data;
      {¬done ∧ ∃v, A2. (p2(v, A2) ∨ p'2(v, A2) ∧ ◇(1)) ∧ ♦(1,2) ∧ (v < e < MAX) ∧ arem(ADD) ∧ (e = E)}
      {¬done ∧ ∃v, A2. (p2(v, A2) ∨ p'2(v, A2)) ∧ ♦(1,2) ∧ ◇(1 + len(A2)) ∧ (v < e < MAX)
       ∧arem(ADD) ∧ (e = E)}
 7    while (u < e) {
 8      p := c;
 9      c := c.next;
10      u := c.data;
11    }
      {¬done ∧ ∃v. (p2(v) ∨ p'2(v) ∧ ◇(1)) ∧ ♦(1,2) ∧ (v < e ≤ u) ∧ arem(ADD) ∧ (e = E)}
12    b := false;
      {¬done ∧ ∃v. (b ∧ (p3(v) ∧ ♦(1,1) ∨ p'3(v) ∧ ♦(1,2) ∧ ◇(1)) ∨ ¬b∧
      (p2(v) ∧ ♦(1,2) ∨ p'2(v) ∧ ♦(1,2) ∧ ◇(1))) ∧  (v < e ≤ u) ∧ arem(ADD) ∧ (e = E)}
      {¬done ∧ ∃v. (b ∧ (p3(v) ∧ ♦(1,1) ∨ p'3(v) ∧ ♦(1,2)) ∨ ¬b∧
      (p2(v) ∧ ♦(1,2) ∨ p'2(v) ∧ ♦(1,2)) ∧ ◇(1)) ∧  (v < e ≤ u) ∧ arem(ADD) ∧ (e = E)}
13    while (!b) {
14      b := cas(p.lock, 0, cid);
15    }
```

Fig. 57. Proof outline of add.

$p_5(v) \stackrel{\text{def}}{=}$
$\exists z, A, A', s, L.\ \mathsf{tmarks}(\mathsf{false}) * \mathsf{ls\_irr}_{\mathsf{cid},L}(\mathsf{Head}, A, \mathsf{p}) * \mathsf{L}_{\mathsf{cid}}(\mathsf{p}, v, \mathsf{c}, \mathsf{false}) * \mathsf{N\_irr}_{\mathsf{cid},s}(\mathsf{c}, \mathsf{u}, z, \mathsf{false})$
$* \mathsf{ls\_irr}_{\mathsf{cid},L}(z, A', \mathtt{null}) * \mathsf{ss}(A :: (v, \mathsf{false}) :: (\mathsf{u}, \mathsf{false}) :: A') * \mathsf{garb}$

$p_6 \stackrel{\text{def}}{=}$
$\exists z, A, v, A', L.\ \mathsf{tmarks}(\mathsf{false}) * \mathsf{ls\_irr}_{\mathsf{cid},L}(\mathsf{Head}, A, \mathsf{p}) * \mathsf{L}_{\mathsf{cid}}(\mathsf{p}, v, z, \mathsf{false})$
$* \mathsf{ls\_irr}_{\mathsf{cid},L}(z, A', \mathtt{null}) * \mathsf{ss}(A :: (v, \mathsf{false}) :: A') * \mathsf{garb}$

$\{\neg\mathsf{done} \wedge \exists v.\ (p_3(v) \wedge \blacklozenge(1,1) \vee p_3'(v) \wedge \blacklozenge(1,2) \wedge \Diamond(1)) \wedge (v < \mathsf{e} \leq \mathsf{u}) \wedge \mathsf{arem}(\mathsf{ADD}) \wedge (\mathsf{e} = \mathsf{E})\}$
16 b := false;
$\{\neg\mathsf{done} \wedge \exists v.\ (\mathsf{b} \wedge (p_4(v) \wedge \blacklozenge(1,0) \vee p_4'(v) \wedge \blacklozenge(1,2) \wedge \Diamond(1))$
$\vee \neg\mathsf{b} \wedge (p_3(v) \wedge \blacklozenge(1,1) \vee p_3'(v) \wedge \blacklozenge(1,2) \wedge \Diamond(1))) \wedge\ (v < \mathsf{e} \leq \mathsf{u}) \wedge \mathsf{arem}(\mathsf{ADD}) \wedge (\mathsf{e} = \mathsf{E})\}$
$\{\neg\mathsf{done} \wedge \exists v.\ (\mathsf{b} \wedge (p_4(v) \wedge \blacklozenge(1,0) \vee p_4'(v) \wedge \blacklozenge(1,2))$
$\vee \neg\mathsf{b} \wedge (p_3(v) \wedge \blacklozenge(1,1) \vee p_3'(v) \wedge \blacklozenge(1,2)) \wedge \Diamond(1)) \wedge\ (v < \mathsf{e} \leq \mathsf{u}) \wedge \mathsf{arem}(\mathsf{ADD}) \wedge (\mathsf{e} = \mathsf{E})\}$
17 while (!b) {
18 b := cas(c.lock, 0, cid);
19 }
$\{\neg\mathsf{done} \wedge \exists v.\ (p_4(v) \wedge \blacklozenge(1,0) \vee p_4'(v) \wedge \blacklozenge(1,2) \wedge \Diamond(1)) \wedge (v < \mathsf{e} \leq \mathsf{u}) \wedge \mathsf{arem}(\mathsf{ADD}) \wedge (\mathsf{e} = \mathsf{E})\}$
20 if (!p.mark && !c.mark && p.next = c) {
21 done := true;
$\{\mathsf{done} \wedge \exists v.\ \mathsf{adjLocked}(\mathsf{p}, v, \mathsf{c}, \mathsf{u}, \mathsf{false}, \mathsf{false}, \mathsf{false}) \wedge \blacklozenge(1,0) \wedge (v < \mathsf{e} \leq \mathsf{u}) \wedge \mathsf{arem}(\mathsf{ADD}) \wedge (\mathsf{e} = \mathsf{E})\}$
22 } else {
$\{\neg\mathsf{done} \wedge \exists v.\ (p_4(v) \vee p_4'(v)) \wedge \blacklozenge(1,2) \wedge \Diamond(1) \wedge (v < \mathsf{e} \leq \mathsf{u}) \wedge \mathsf{arem}(\mathsf{ADD}) \wedge (\mathsf{e} = \mathsf{E})\}$
23 p.lock := 0;
24 c.lock := 0;
$\{\neg\mathsf{done} \wedge \exists v.\ (p_2(v) \vee p_2'(v)) \wedge \blacklozenge(1,2) \wedge \Diamond(1) \wedge (v < \mathsf{e} \leq \mathsf{u}) \wedge \mathsf{arem}(\mathsf{ADD}) \wedge (\mathsf{e} = \mathsf{E})\}$
25 }
26 }
$\{\exists v.\ \mathsf{adjLocked}(\mathsf{p}, v, \mathsf{c}, \mathsf{u}, \mathsf{false}, \mathsf{false}, \mathsf{false}) \wedge \blacklozenge(1,0) \wedge (v < \mathsf{e} \leq \mathsf{u}) \wedge \mathsf{arem}(\mathsf{ADD}) \wedge (\mathsf{e} = \mathsf{E})\}$
27 c.lock := 0;
$\{\exists v.\ p_5(v) \wedge \blacklozenge(1,0) \wedge (v < \mathsf{e} \leq \mathsf{u}) \wedge \mathsf{arem}(\mathsf{ADD}) \wedge (\mathsf{e} = \mathsf{E})\}$
28 if (u != e) {
$\{\exists v.\ p_5(v) \wedge \blacklozenge(1,0) \wedge (v < \mathsf{e} < \mathsf{u}) \wedge \mathsf{arem}(\mathsf{ADD}) \wedge (\mathsf{e} = \mathsf{E})\}$
29 x := cons(0, e, c, false);
30 p.next := x;
31 r := true;
$\{p_6 \wedge \mathsf{arem}(\mathbf{skip}) \wedge (\mathsf{r} = \mathsf{R})\}$
32 } else {
$\{\exists v.\ p_5(v) \wedge \blacklozenge(1,0) \wedge (\mathsf{e} = \mathsf{u}) \wedge \mathsf{arem}(\mathsf{ADD}) \wedge (\mathsf{e} = \mathsf{E})\}$
33 r := false;
$\{p_6 \wedge \mathsf{arem}(\mathbf{skip}) \wedge (\mathsf{r} = \mathsf{R})\}$
34 }
$\{p_6 \wedge \mathsf{arem}(\mathbf{skip}) \wedge (\mathsf{r} = \mathsf{R})\}$
35 p.lock := 0;
$\{P \wedge \mathsf{arem}(\mathbf{skip}) \wedge (\mathsf{r} = \mathsf{R})\}$
36 return r;
}

Fig. 58. Proof outline of 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;
      {p6 ∧ 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;
      {p6 ∧ arem(skip) ∧ (r = R)}
34  }
    {p6 ∧ arem(skip) ∧ (r = R)}
35  p.lock := 0;
36  c.lock := 0;
    {P ∧ arem(skip) ∧ (r = R)}
37  return r;
}
```

Fig. 59. Proof outline of rmv.

C.6 Transfer lists with TAS lock

Fig. 60 and 61 shows the implementation of the simplified transfer lists, where we assume there are only two lists l1 and l2. And for convenience, we use subscript in the method to distinguish the list that the method operates on (e.g., $\texttt{locate}_1(e)$ searches on l1). The abstract operation for each method is defined in Fig. 62. Note that MOVE should atomically transfer the head of $\texttt{S}_1$ to $\texttt{S}_2$.

To verify the example, we introduce some auxiliary variables. The local variable aux and shared variable toadd have the same functionality as in lock coupling lists. The shared variable todel is an option value of a pair of list and value. It represents a temporarily removed node in the move method (see line 28 in Fig. 61). When move first removes the node in the source list, the concrete list does not match the abstract representation because the abstract operation has not taken effect. So we introduce todel to help establish the relation between the concrete list and abstract representation. The auxiliary variable level in each list helps give their dependencies.

Fig. 64 gives the invariants. Most part of the invariants are the same as lock coupling lists, except we have some extra properties for move method. When the lock mutex is not holden, we know that the levels of the lists are both 1 and todel is None, because these variables only change in move with mutex holden and get restored before release mutex. Fig. 65 shows the rely and guarantee conditions. We have *MoveRmv* and *Move* to respectively represent the remove and add action of the move. There are also the pair of *Lockmutex* and *Unockmutex* for acquiring and releasing the lock mutex, and a *SetLevel* for setting the level of list in move method.

Fig. 66 gives the definite actions and layers. As we have introduced in Sec. 5, the layer is a pair of *Nat* (n_1,n_2). n_1 gives the dependency relation among lists, which corresponds to the level field. n_2 represents the dependency relation in a list, decided based on the node's distance to the end of the list. We can flexiblely alter the dependencies among lists by setting the level field.

The proof of move is given in Fig. 67 and Fig. 68, while the proofs for add and rmv are similar to the lock coupling lists and omitted here. move may encounter a long dependency chain as it has to traverse two lists. However, it is not a problem for us because the layered definite actions can allow modular and local progress proofs on each lock.

```
bool[] toadd; //initially all false
int mutex;
struct Node {
  int lock;
  int data;
  struct Node *next;
}
struct List {
  struct Node *Head;
  int level;
}
```

```
initialize(){
  l1.Head := cons(0, GUARD, null);
  l1.Head.next := cons(0, GUARD, null);
  l2.Head := cons(0, GUARD, null);
  l2.Head.next := cons(0, GUARD, null);
  l1.level := 1;
  l2.level := 1;
  mutex := 0;
  todel := None;
}
```

```
locate1(e):
 1  local p, c, u, aux;
 2  p := l1.Head;
 3  lock(p);
 4  c := p.next;
 5  <u := c.data; aux := metricTAS(p)>;
 6  while (u != e && u != GUARD) {
 7    lock(c);
 8    unlock(p);
 9    p := c;
10    c := p.next;
11    <u := c.data; aux := metricTAS(p)>;
12  }
13  return (p,c);
add1(e):
14  local x, y, z, u;
15  toadd_cid := true;
16  (x, z) := locate1(e);
17  u := z.data;
18  if (u = GUARD) {
19    y := cons(0, e, z);
20    <x.next := y; toadd_cid := false>;
21    r := true;
22  } else {
23    <r := false;toadd_cid := false>;
24  }
25  unlock(x);
26  return r;
rmv1(e):
27  local x, y, z, v;
28  (x, y) := locate1(e);
29  v := y.data;
30  if (v = e) {
31    lock(y);
32    z := y.next;
33    x.next := z;
34    unlock(x);
35    unlock(y);
36    dispose(y);
37    return true;
38  } else {
39    unlock(x);
40    return false;
41  }
```

Fig. 60. Code for transfer lists with TAS lock (except move method).

```
move_12():
 1  local x, y, z, e, m, n, u;
 2  lock(mutex);
 3  toadd_cid := true;
 4  x := l1.head;
 5  lock(x);
 6  y := x.next;
 7  e := y.data;
 8  if (e = GUARD) {
      // l1 is empty
 9    unlock(x);
10    unlock(mutex);
11    toadd_cid := false;
12    return false;
13  }
14  <l2.level := 0;>
15  (m, n) := locate(l2,e);
16  u := n.data;
17  if (u = e) {
      // if e already in l2
18    unlock(x);
19    unlock(m);
20    <l2.level := 1;>
21    unlock(mutex);
22    toadd_cid := false;
23    return false
24  }
    // remove y in l1
25  <l2.level := 2;>
26  lock(y);
27  z := y.next;
28  <x.next := z; todel := Some (l1, e);>
    // insert y at the end of l2
29  y.next := m.next;
30  <m.next := y;
    toadd_cid := false; todel := None;>
31  unlock(x);
32  unlock(y);
33  unlock(m);
34  <l2.level := 1;>
35  unlock(mutex);
36  return true;
```

Fig. 61. Code for move method of transfer lists with TAS lock.

```
ADD(S, E) {
  local R;
  < if (In E S) {
      R := false;
    } else {
      S := S :: E;
      R := true;
    }
  >
  return R;
}
```

```
RMV(S, E) {
  local R;
  < if (~ In E S) {
      R := false;
    } else {
      S := remove(S, E);
      R := true;
    }
  >
  return R;
}
```

```
MOVE(S1, S2) {
  local R;
  < if (S1 = nil ∨ In S1.head S2) {
      R := false;
    } else {
      S1 := removehead(S1);
      S2 := S2 :: S1.head;
      R := true;
    }
  >
  return R;
}
```

Fig. 62. Abstract operations for transfer lists.

```
int metricTAS(p){
 1  local n, l, ts, o, t;
 2  n := p.next; l := 0; ts := ∅;
 3  while (n <> null) {
 4    l++;
 5    t := n.lock;
 6    if (t != 0) {
 7      if (toadd_t = true) ts := ts ∪ {t};
 8    }
 9    n := n.next;
10  }
11  l := l + |ts|;
12  return l;
}
```

$$|\emptyset| \stackrel{\text{def}}{=} 0 \qquad |S \cup \{x\}| \stackrel{\text{def}}{=} |S| + 1$$

$$\mathsf{len}(\epsilon) \stackrel{\text{def}}{=} 0 \qquad \mathsf{len}(v :: A) \stackrel{\text{def}}{=} \mathsf{len}(A) + 1$$

$$\mathsf{toaddThrds}(S) \stackrel{\text{def}}{=} \{\mathsf{t} \mid (\mathsf{t} \in S) \wedge (\mathsf{toadd}_\mathsf{t} = \mathsf{true})\}$$

Fig. 63. Auxiliary code to compute the metric for the list traversal.

$A ::= \epsilon \mid v :: A \qquad s ::= 0 \mid \mathsf{t} \qquad L ::= \epsilon \mid s :: L \qquad B \in \mathit{ThrdID} \rightharpoonup \mathit{Bool}$

$I \stackrel{\text{def}}{=}$
$\exists A_1, A_2, L_1, L_2.$
$\mathsf{ls}_{L_1}(l_1.\mathtt{Head}, A_1, \mathtt{null}) * \mathsf{ss}(A_1, S_1, \mathtt{todel})$
$* \mathsf{ls}_{L_2}(l_2.\mathtt{Head}, A_2, \mathtt{null}) * \mathsf{ss}(A_2, S_2, \mathtt{todel}) * \mathsf{mutex\_lock} * \mathsf{toadds} * \mathsf{levels} \wedge \mathsf{props}$

$\mathsf{props} \stackrel{\text{def}}{=} (\mathit{mutex} = 0 \rightarrow (l1.\mathtt{level} = 1 \wedge l2.\mathtt{level} = 1 \wedge \mathit{todel} = \mathit{None}))$

$\mathsf{toadds} \stackrel{\text{def}}{=} \exists B.\ \mathsf{toadds}(B) \qquad \mathsf{toadds}(B) \stackrel{\text{def}}{=} (\circledast_{\mathsf{t}} \mathsf{toadd}_{\mathsf{t}} = B(\mathsf{t}))$

$\mathsf{toadd}_{\mathsf{t}}(b) \stackrel{\text{def}}{=} \exists B.\ \mathsf{toadd}_{\mathsf{t}}(b, B) \qquad \mathsf{toadd}_{\mathsf{t}}(b, B) \stackrel{\text{def}}{=} (\mathsf{toadd}_{\mathsf{t}} = b) * (\circledast_{\mathsf{t}' \neq \mathsf{t}} \mathsf{toadd}_{\mathsf{t}'} = B(\mathsf{t}'))$

$\mathsf{levels} \stackrel{\text{def}}{=} (l_1.\mathtt{level} = \_) * (l_2.\mathtt{level} = \_) \qquad \mathsf{levels(lev1,lev2)} \stackrel{\text{def}}{=} (l_1.\mathtt{level} = \mathit{lev1}) * (l_2.\mathtt{level} = \mathit{lev2})$

$\mathsf{ls}_L(x, A, y) \stackrel{\text{def}}{=}$
$(x = y \wedge A = \epsilon \wedge L = \epsilon \wedge \mathsf{emp})$
$\vee\ (x \neq y \wedge \exists z, v, A', s, L'.\ A = v :: A' \wedge L = s :: L' \wedge \mathsf{N}_s(x, v, z) * \mathsf{ls}_{L'}(z, A', y))$

$\mathsf{ls\_irr}_{\mathsf{t},L}(x, A, y) \stackrel{\text{def}}{=}$
$(x = y \wedge A = \epsilon \wedge L = \epsilon \wedge \mathsf{emp})$
$\vee\ (x \neq y \wedge \exists z, v, A', s, L'.\ A = v :: A' \wedge L = s :: L' \wedge \mathsf{N\_irr}_{\mathsf{t},s}(x, v, z) * \mathsf{ls\_irr}_{\mathsf{t},L'}(z, A', y))$

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

$\mathsf{N}_s(p, v, y) \stackrel{\text{def}}{=} \mathsf{lock}_s(p) * (p.\mathtt{data} = v) * (p.\mathtt{next} = y)$

$\mathsf{N\_irr}_{\mathsf{t},s}(p, v, y) \stackrel{\text{def}}{=} \mathsf{lock\_irr}_{\mathsf{t},s}(p) * (p.\mathtt{data} = v) * (p.\mathtt{next} = y)$

$\mathsf{L}_s(p, v, y) \stackrel{\text{def}}{=} \mathsf{locked}_s(p) * (p.\mathtt{data} = v) * (p.\mathtt{next} = y)$

$\mathsf{L\_irr}_{\mathsf{t},s}(p, v, y) \stackrel{\text{def}}{=} (s \neq \mathsf{t}) \wedge \mathsf{L}_s(p, v, y)$

$\mathsf{U}(p, v, y) \stackrel{\text{def}}{=} \mathsf{unlocked}(p) * (p.\mathtt{data} = v) * (p.\mathtt{next} = y)$

$\mathsf{lock}_s(p) \stackrel{\text{def}}{=} \mathsf{locked}_s(p) \vee (s = 0 \wedge \mathsf{unlocked}(p)) \qquad \mathsf{lock\_irr}_{\mathsf{t},s}(p) \stackrel{\text{def}}{=} \mathsf{lock}_s(p) \wedge (\mathsf{t} \neq s)$

$\mathsf{locked}_{\mathsf{t}}(p) \stackrel{\text{def}}{=} (p.\mathtt{lock} = \mathsf{t}) \wedge (\mathsf{t} \in \mathtt{TIDS}) \qquad \mathsf{unlocked}(p) \stackrel{\text{def}}{=} (p.\mathtt{lock} = 0)$

$\mathsf{mutex\_lock} \stackrel{\text{def}}{=} (\mathtt{mutex} = \_) \qquad \mathsf{mutex\_locked}_{\mathsf{t}} \stackrel{\text{def}}{=} \mathtt{mutex} = \mathsf{t} \wedge \mathsf{t} \in \mathtt{TIDS}$

$\mathsf{mutex\_unlocked} \stackrel{\text{def}}{=} \mathtt{mutex} = 0 \qquad \mathsf{mutex\_irr}_{\mathsf{t}} \stackrel{\text{def}}{=} \exists \mathsf{t}', \mathtt{mutex} = \mathsf{t}' \wedge \mathsf{t}' \neq \mathsf{t}$

$\mathsf{s}(A) \stackrel{\text{def}}{=} \exists A'.\ (A = \mathtt{GUARD} :: A' :: \mathtt{GUARD})$

$\mathsf{ss}(A, S, \mathit{todel}) \stackrel{\text{def}}{=}$
$\exists A', l, v.\ (A = \mathtt{GUARD} :: A' :: \mathtt{GUARD}) \wedge$
$((\mathsf{map}(S, A', l, v) * \mathit{todel} = \mathit{Some}(l, v)) \vee (S \Longmapsto A' * \mathit{todel} = \mathit{None}))$

$\mathsf{map}(S, A', l, v) \stackrel{\text{def}}{=} (\mathsf{match}(S, l) \wedge S \Longmapsto A' :: v) \vee (\neg \mathsf{match}(S, l) \wedge S \Longmapsto A')$

$\mathsf{match}(S, l) \stackrel{\text{def}}{=} (S = S_1 \wedge l = l_1) \vee (S = S_2 \wedge l = l_2)$

Fig. 64. Invariant and precondition of the transfer lists with TAS lock.

$$
\begin{aligned}
&G_{\mathtt{cid}} \stackrel{\text{def}}{=} \\
&\quad (\mathit{ToAdd}_{\mathtt{cid}} \vee \mathit{Add}_{\mathtt{cid}} \vee \mathit{FailedAdd}_{\mathtt{cid}} \vee \mathit{Move}_{\mathtt{cid}} \vee \mathit{Rmv}_{\mathtt{cid}} \vee \mathit{MoveRmv}_{\mathtt{cid}} \vee \mathit{Unlockmutex}_{\mathtt{cid}} \\
&\quad \vee \mathit{Lockmutex}_{\mathtt{cid}} \vee \mathit{LockH}_{\mathtt{cid}} \vee \mathit{Lock2}_{\mathtt{cid}} \vee \mathit{Unlock}_{\mathtt{cid}} \vee \mathit{SetLevel}_{\mathtt{cid}} \vee \mathsf{Id}) * \mathsf{Id} \wedge (I \ltimes I)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{ToAdd}_{\mathsf{t}} \stackrel{\text{def}}{=} \exists A_1, A_2, L_1, L_2.\ [\mathsf{ls\_irr}_{\mathsf{t},L_1}(l_1.\mathsf{Head}, A_1, \mathtt{null})] * [\mathsf{ls\_irr}_{\mathsf{t},L_2}(l_2.\mathsf{Head}, A_2, \mathtt{null})] \\
&\qquad *((\mathtt{toadd}_{\mathsf{t}} = \mathsf{false}) \ltimes (\mathtt{toadd}_{\mathsf{t}} = \mathsf{true}))
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{Add}_{\mathsf{t}} \stackrel{\text{def}}{=} \\
&\quad \exists x, y, z, v, w, l, A, L, \mathsf{S}, S. \\
&\quad [\mathsf{ls\_irr}_{\mathsf{t},L}(l.\mathsf{Head}, A, x)] * \\
&\quad ((\mathsf{L}_{\mathsf{t}}(x, v, z) * (\mathtt{toadd}_{\mathsf{t}} = \mathsf{true}) * (S \Longmapsto \mathsf{S}) \wedge \mathsf{match}(S, l)) \ltimes \\
&\quad (\mathsf{L}_{\mathsf{t}}(x, v, y) * \mathsf{U}(y, w, z) * (\mathtt{toadd}_{\mathsf{t}} = \mathsf{false}) * (S \Longmapsto \mathsf{S} :: w) \wedge w \neq \mathtt{GUARD}))
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{Move}_{\mathsf{t}} \stackrel{\text{def}}{=} \\
&\quad \exists x, y, z, v, w, l, l', A, L, \mathsf{S}, S, \mathsf{S}', S'. \\
&\quad [\mathsf{ls\_irr}_{\mathsf{t},L}(l.\mathsf{Head}, A, x)] * [\mathsf{mutex\_locked}_{\mathsf{t}}] * \\
&\quad ((\mathsf{L}_{\mathsf{t}}(x, v, z) * (\mathtt{toadd}_{\mathsf{t}} = \mathsf{true}) * (\mathtt{todel} = \mathit{Some}(l', w)) \\
&\quad *(S \Longmapsto \mathsf{S}) * (S' \Longmapsto w :: \mathsf{S}') \wedge \mathsf{match}(S, l) \wedge \mathsf{match}(S', l')) \ltimes \\
&\quad (\mathsf{L}_{\mathsf{t}}(x, v, y) * \mathsf{L}_{\mathsf{t}}(y, w, z) * (\mathtt{toadd}_{\mathsf{t}} = \mathsf{false}) * (\mathtt{todel} = \mathit{None}) * (S \Longmapsto \mathsf{S} :: w) * (S' \Longmapsto \mathsf{S}')))
\end{aligned}
$$

$$
\mathit{FailedAdd}_{\mathsf{t}} \stackrel{\text{def}}{=} (\mathtt{toadd}_{\mathsf{t}} = \mathsf{true}) \ltimes (\mathtt{toadd}_{\mathsf{t}} = \mathsf{false})
$$

$$
\begin{aligned}
&\mathit{Rmv}_{\mathsf{t}} \stackrel{\text{def}}{=} \\
&\quad \exists x, y, z, v, u, l, A, L, \mathsf{S}, S. \\
&\quad [\mathsf{ls\_irr}_{\mathsf{t},L}(l.\mathsf{Head}, A, x)] * \\
&\quad (\mathsf{L}_{\mathsf{t}}(x, v, y) * \mathsf{L}_{\mathsf{t}}(y, u, z) * (S \Longmapsto \mathsf{S}) \wedge \mathsf{match}(S, l)) \ltimes (\mathsf{L}_{\mathsf{t}}(x, v, z) * (S \Longmapsto \mathit{remove}(\mathsf{S}, u)))
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{MoveRmv}_{\mathsf{t}} \stackrel{\text{def}}{=} \\
&\quad \exists x, y, z, v, u, l, A, L. \\
&\quad [\mathsf{ls\_irr}_{\mathsf{t},L}(l.\mathsf{Head}, A, x)] * [\mathsf{mutex\_locked}_{\mathsf{t}}] * \\
&\quad (\mathsf{L}_{\mathsf{t}}(x, v, y) * \mathsf{L}_{\mathsf{t}}(y, u, z) * \mathit{todel} = \mathit{None}) \ltimes (\mathsf{L}_{\mathsf{t}}(x, v, z) * \mathit{todel} = \mathit{Some}(l, u))
\end{aligned}
$$

$$
\mathit{Unlockmutex}_{\mathtt{cid}} \stackrel{\text{def}}{=} \mathsf{mutex\_locked}_{\mathsf{t}} \ltimes_1 \mathsf{mutex\_unlocked}
$$

$$
\begin{aligned}
&\mathit{Lockmutex}_{\mathsf{t}} \stackrel{\text{def}}{=} \\
&\quad \exists A, B, L_1, L_2.\ (\mathsf{mutex\_unlocked} \ltimes_1 \mathsf{mutex\_locked}) * \\
&\quad [\mathsf{ls\_irr}_{\mathsf{t},L_1}(l_1.\mathsf{Head}, A, \mathtt{null})] * [\mathsf{ls\_irr}_{\mathsf{t},L_2}(l_2.\mathsf{Head}, B, \mathtt{null})]
\end{aligned}
$$

$$
\mathit{LockH}_{\mathsf{t}} \stackrel{\text{def}}{=} \exists x, A, l, L.\ (\mathsf{U}(l.\mathsf{Head}, \mathtt{GUARD}, x) \ltimes_1 \mathsf{L}_{\mathsf{t}}(l.\mathsf{Head}, \mathtt{GUARD}, x)) * [\mathsf{ls\_irr}_{\mathsf{t},L}(x, A, \mathtt{null})]
$$

$$
\begin{aligned}
&\mathit{Lock2}_{\mathsf{t}} \stackrel{\text{def}}{=} \\
&\quad \exists x, y, z, A, v, u, A', l, L, L'.\ [\mathsf{ls\_irr}_{\mathsf{t},L}(l.\mathsf{Head}, A, x) * \mathsf{L}_{\mathsf{t}}(x, v, y)] * (\mathsf{U}(y, u, z) \ltimes \mathsf{L}_{\mathsf{t}}(y, u, z)) \\
&\quad * [\mathsf{ls\_irr}_{\mathsf{t},L'}(z, A', \mathtt{null})]
\end{aligned}
$$

$$
\mathit{Unlock}_{\mathsf{t}} \stackrel{\text{def}}{=} \exists x, y, A, v, l, L.\ [\mathsf{ls\_irr}_{\mathsf{t},L}(l.\mathsf{Head}, A, x)] * (\mathsf{L}_{\mathsf{t}}(x, v, y) \ltimes \mathsf{U}(x, v, y))
$$

$$
\mathit{SetLevel}_{\mathsf{t}} \stackrel{\text{def}}{=} \exists l.\ [\mathsf{mutex\_locked}_{\mathsf{t}}] * ((l.\mathtt{level} = \_) \ltimes (l.\mathtt{level} = \_))
$$

Fig. 65. Rely and guarantee of the transfer lists with TAS lock.

$\mathcal{D}_\mathsf{t} \stackrel{\text{def}}{=} \mathcal{B}1_\mathsf{t} \wedge (\forall \mathsf{x}.\, \mathcal{B}2_\mathsf{t}(\mathsf{x})) \wedge \mathcal{B}_0 \qquad \mathit{LayFeat} \stackrel{\text{def}}{=} \mathit{Addr} \mid \mathcal{M} \mid \bot$

$\mathcal{B}_0 \stackrel{\text{def}}{=} (\mathbf{false} \rightsquigarrow \mathbf{true}, \bot)$

$\mathcal{B}1_\mathsf{t} \stackrel{\text{def}}{=} (\mathsf{bp1}_\mathsf{t} \rightsquigarrow \mathsf{bq1}, \mathcal{M})$

$\mathsf{bp1}_\mathsf{t} \stackrel{\text{def}}{=} \mathsf{mutex\_locked}_\mathsf{t} * \mathbf{true} \wedge I \qquad \mathsf{bq1} \stackrel{\text{def}}{=} \mathsf{mutex\_unlocked} * \mathbf{true} \wedge I$

$\mathcal{B}2_\mathsf{t}(\mathsf{x}) \stackrel{\text{def}}{=} (\mathsf{bp2}_\mathsf{t}(\mathsf{x}) \rightsquigarrow \mathsf{bq2}(\mathsf{x}), \mathsf{x})$

$\mathsf{bp2}_\mathsf{t}(\mathsf{x}) \stackrel{\text{def}}{=} \mathsf{locked}_\mathsf{t}(\mathsf{x}) * \mathbf{true} \wedge I \qquad \mathsf{bq2}(\mathsf{x}) \stackrel{\text{def}}{=} \mathsf{unlocked}(\mathsf{x}) * \mathbf{true} \wedge I$

$\mathit{Layer} \stackrel{\text{def}}{=} \mathit{Nat} \times \mathit{Nat} \qquad (\mathsf{n}_1, \mathsf{n}_2) < (\mathsf{n}_1', \mathsf{n}_2') \text{ iff } \mathsf{n}_1 < \mathsf{n}_1' \vee (\mathsf{n}_1 = \mathsf{n}_1' \wedge \mathsf{n}_2 < \mathsf{n}_2')$

$$\mathfrak{L}(\mathfrak{S}, \mathcal{B}) = \begin{cases} (2,0) & \text{if } \mathcal{B}.\mathcal{F} = \mathcal{M} \\ (\mathsf{n}, \mathsf{len(B)}) & \text{if } \mathcal{B}.\mathcal{F} = \mathsf{x} \wedge \mathfrak{S} \models \mathsf{P} \\ (0,0) & \text{otherwise} \end{cases}$$

$\mathsf{P} \stackrel{\text{def}}{=} \exists L, L', \mathsf{l}, \mathsf{A}, \mathsf{B}, \mathsf{n}.\, \mathsf{ls}_L(\mathtt{l.Head}, \mathsf{A}, \mathsf{x}) * \mathsf{ls}_{L'}(\mathsf{x}, \mathsf{B}, \mathtt{null}) * (\mathtt{l.level} = \mathsf{n}) * \mathbf{true} \wedge I$

Fig. 66. Definite actions and layers of the transfer lists.

$\mathsf{ls\_irr}_{cid}(todel, head, S) \stackrel{\text{def}}{=}$
$\quad \exists A_1, L_1.\mathsf{ls\_irr}_{cid,L_1}(head, A_1, \mathtt{null}) * \mathsf{ss}(A_1, S, todel)$
$\mathsf{adjacent}_{\mathtt{cid}}(todel, head, S, x, v, y, u, z) \stackrel{\text{def}}{=}$
$\quad \exists A_1, A_2, L_1, L_2, s.\ \mathsf{ls\_irr}_{cid,L_1}(head, A_1, x) * \mathsf{L}_{\mathtt{cid}}(\mathsf{x}, v, \mathsf{y}) * \mathsf{N\_irr}_{cid,s}(\mathsf{y}, \mathsf{u}, z) *$
$\quad \mathsf{ls\_irr}_{\mathtt{cid},L_2}(z, A_2, \mathtt{null}) * \mathsf{ss}(A_1 :: v :: \mathsf{u} :: A_2, S, todel)$
$\mathsf{p1}_{\mathtt{cid}}(b, lev1, lev2, tk, aop) \stackrel{\text{def}}{=}$
$\quad \mathsf{mutex\_irr}_{\mathtt{cid}} * \mathsf{toadd}_{\mathtt{cid}}(b) * \mathsf{levels}(lev1, lev2) \wedge \mathsf{props} \wedge \blacklozenge(tk) \wedge \mathsf{arem}(aop)$
$\mathsf{p2}_{\mathtt{cid}}(b, lev1, lev2, tk, aop) \stackrel{\text{def}}{=}$
$\quad \mathsf{mutex\_locked}_{\mathtt{cid}} * \mathsf{toadd}_{\mathtt{cid}}(b) * \mathsf{levels}(lev1, lev2) \wedge \mathsf{props} \wedge \blacklozenge(tk) \wedge \mathsf{arem}(aop)$

$\mathtt{move}_{12}$():

```
 1  local x, y, z, e, m, n, u;
```
$\{\exists todel.\ \mathsf{ls\_irr}_{\mathtt{cid}}(todel, l1.\mathsf{Head}, S_1) * \mathsf{ls\_irr}_{\mathtt{cid}}(todel, l2.\mathsf{Head}, S_2) * \mathsf{p1}_{\mathtt{cid}}(false, \_, \_, 3, MOVE)\}$
```
 2  lock(mutex);
```
$\{\mathsf{ls\_irr}_{\mathtt{cid}}(None, l1.\mathsf{Head}, S_1) * \mathsf{ls\_irr}_{\mathtt{cid}}(None, l2.\mathsf{Head}, S_2) * \mathsf{p2}_{\mathtt{cid}}(false, 1, 1, 2, MOVE)\}$
```
 3  toadd_cid := true;
```
$\{\mathsf{ls\_irr}_{\mathtt{cid}}(None, l1.\mathsf{Head}, S_1) * \mathsf{ls\_irr}_{\mathtt{cid}}(None, l2.\mathsf{Head}, S_2) * \mathsf{p2}_{\mathtt{cid}}(true, 1, 1, 2, MOVE)\}$
```
 4  x := l1.head;
```
$\{\mathsf{ls\_irr}_{\mathtt{cid}}(None, l1.\mathsf{Head}, S_1) * \mathsf{ls\_irr}_{\mathtt{cid}}(None, l2.\mathsf{Head}, S_2) * \mathsf{p2}_{\mathtt{cid}}(true, 1, 1, 2, MOVE) \wedge x = l1.Head\}$
```
 5  lock(x);
```
$\{\exists y, e, z.\ \mathsf{adjacent}_{\mathtt{cid}}(None, l1.\mathsf{Head}, S_1, x, GUARD, y, e, z) * \mathsf{ls\_irr}_{\mathtt{cid}}(None, l2.\mathsf{Head}, S_2)$
$\ *\mathsf{p2}_{\mathtt{cid}}(true, 1, 1, 1, MOVE) \wedge x = l1.Head\}$
```
 6  y := x.next;
```
$\{\exists e, z.\ \mathsf{adjacent}_{\mathtt{cid}}(None, l1.\mathsf{Head}, S_1, x, GUARD, y, e, z) * \mathsf{ls\_irr}_{\mathtt{cid}}(None, l2.\mathsf{Head}, S_2)$
$\ *\mathsf{p2}_{\mathtt{cid}}(true, 1, 1, 1, MOVE) \wedge x = l1.Head\}$
```
 7  e := y.data;
```
$\{\exists z.\ \mathsf{adjacent}_{\mathtt{cid}}(None, l1.\mathsf{Head}, S_1, x, GUARD, y, e, z) * \mathsf{ls\_irr}_{\mathtt{cid}}(None, l2.\mathsf{Head}, S_2)$
$\ *\mathsf{p2}_{\mathtt{cid}}(true, 1, 1, 1, MOVE) \wedge x = l1.Head\}$
```
 8  if(e=GUARD)
```
$\{\exists z.\ \mathsf{adjacent}_{\mathtt{cid}}(None, l1.\mathsf{Head}, S_1, x, GUARD, y, e, z) * \mathsf{ls\_irr}_{\mathtt{cid}}(None, l2.\mathsf{Head}, S_2)$
$\ *\mathsf{p2}_{\mathtt{cid}}(true, 1, 1, 1, MOVE) \wedge x = l1.Head \wedge e = GUARD\}$
```
 9    unlock(x);
10    unlock(mutex);
11    toadd_cid := false;
```
$\{\exists todel.\ \mathsf{ls\_irr}_{\mathtt{cid}}(todel, l1.\mathsf{Head}, S_1) * \mathsf{ls\_irr}_{\mathtt{cid}}(todel, l2.\mathsf{Head}, S_2) * \mathsf{p1}_{\mathtt{cid}}(false, \_, \_, 1, skip)\}$
```
12    return false;
13  }
```
$\{\exists z.\ \mathsf{adjacent}_{\mathtt{cid}}(None, l1.\mathsf{Head}, S_1, x, GUARD, y, e, z) * \mathsf{ls\_irr}_{\mathtt{cid}}(None, l2.\mathsf{Head}, S_2)$
$\ *\mathsf{p2}_{\mathtt{cid}}(true, 1, 1, 1, MOVE) \wedge x = l1.Head \wedge e \neq GUARD\}$
```
14  <l2.level := 0;>
```
$\{\exists z.\ \mathsf{adjacent}_{\mathtt{cid}}(None, l1.\mathsf{Head}, S_1, x, GUARD, y, e, z) * \mathsf{ls\_irr}_{\mathtt{cid}}(None, l2.\mathsf{Head}, S_2)$
$\ *\mathsf{p2}_{\mathtt{cid}}(true, 1, 0, 1, MOVE) \wedge x = l1.Head \wedge e \neq GUARD\}$

Fig. 67. Proof outline of move method (1).

$\mathsf{adjacent\_notin}_{\mathsf{cid}}(todel, head, S, x, v, y, u, z, e) \stackrel{\text{def}}{=}$
$\exists A_1, A_2, L_1, L_2, s.\ \mathsf{ls\_irr}_{cid,L_1}(head, A_1, x) * \mathsf{L}_{\mathsf{cid}}(\mathsf{x}, v, \mathsf{y}) * \mathsf{N\_irr}_{cid,s}(\mathsf{y}, \mathsf{u}, z) *$
$\mathsf{ls\_irr}_{\mathsf{cid},L_2}(z, A_2, \mathtt{null}) * \mathsf{ss}(A_1 :: v :: \mathsf{u} :: A_2, S, todel) \wedge \neg In\ e\ (A_1 :: v)$

$\mathsf{adjacent\_locked}_{\mathsf{cid}}(todel, head, S, x, v, y, u, z) \stackrel{\text{def}}{=}$
$\exists A_1, A_2, L_1, L_2.\ \mathsf{ls\_irr}_{cid,L_1}(head, A_1, x) * \mathsf{L}_{\mathsf{cid}}(\mathsf{x}, v, \mathsf{y}) * \mathsf{L}_{\mathsf{cid}}(\mathsf{y}, \mathsf{u}, z) *$
$\mathsf{ls\_irr}_{\mathsf{cid},L_2}(z, A_2, \mathtt{null}) * \mathsf{ss}(A_1 :: v :: \mathsf{u} :: A_2, S, todel)$

```
15  (m,n) := locate(l2, e);
```
$\{\exists z, c, u, p.\ \mathsf{adjacent}_{\mathsf{cid}}(None, l1.\mathsf{Head}, S_1, x, GUARD, y, e, z) * \mathsf{adjacent\_notin}_{\mathsf{cid}}(None, l2.\mathsf{Head}, S_2, m, c, n, u, p, e)$
$* \mathsf{p2}_{\mathsf{cid}}(true, 1, 0, 0, MOVE) \wedge x = l1.Head \wedge e \neq GUARD \wedge (e \neq u \rightarrow p = \mathtt{null})\}$
```
16  u := n.data;
```
$\{\exists z, c, p.\ \mathsf{adjacent}_{\mathsf{cid}}(None, l1.\mathsf{Head}, S_1, x, GUARD, y, e, z) * \mathsf{adjacent\_notin}_{\mathsf{cid}}(None, l2.\mathsf{Head}, S_2, m, c, n, u, p, e)$
$* \mathsf{p2}_{\mathsf{cid}}(true, 1, 0, 0, MOVE) \wedge x = l1.Head \wedge e \neq GUARD \wedge (e \neq u \rightarrow p = \mathtt{null})\}$
```
17  if(u=e) {
18    unlock(x);
19    unlock(m);
20    <l2.level := 1;>
21    unlock(mutex);
22    toadd_cid := false;
```
$\{\exists todel.\ \mathsf{ls\_irr}_{\mathsf{cid}}(todel, l1.\mathsf{Head}, S_1) * \mathsf{ls\_irr}_{\mathsf{cid}}(todel, l2.\mathsf{Head}, S_2) * \mathsf{p1}_{\mathsf{cid}}(false, \_, \_, 0, skip)\}$
```
23    return false;
24  }
```
$\{\exists z, c.\ \mathsf{adjacent}_{\mathsf{cid}}(None, l1.\mathsf{Head}, S_1, x, GUARD, y, e, z) * \mathsf{adjacent\_notin}_{\mathsf{cid}}(None, l2.\mathsf{Head}, S_2, m, c, n, u, \mathtt{null}, e)$
$* \mathsf{p2}_{\mathsf{cid}}(true, 1, 0, 0, MOVE) \wedge x = l1.Head \wedge e \neq GUARD \wedge e \neq u\}$
```
25  <l2.level := 2;>
```
$\{\exists z, c.\ \mathsf{adjacent}_{\mathsf{cid}}(None, l1.\mathsf{Head}, S_1, x, GUARD, y, e, z) * \mathsf{adjacent\_notin}_{\mathsf{cid}}(None, l2.\mathsf{Head}, S_2, m, c, n, u, \mathtt{null}, e)$
$* \mathsf{p2}_{\mathsf{cid}}(true, 1, 2, 0, MOVE) \wedge x = l1.Head \wedge e \neq GUARD \wedge e \neq u\}$
```
26  lock(y);
```
$\{\exists z, c.\ \mathsf{adjacent\_locked}_{\mathsf{cid}}(None, l1.\mathsf{Head}, S_1, x, GUARD, y, e, z) * \mathsf{adjacent\_notin}_{\mathsf{cid}}(None, l2.\mathsf{Head}, S_2, m, c, n, u, \mathtt{null},$
$* \mathsf{p2}_{\mathsf{cid}}(true, 1, 2, 0, MOVE) \wedge x = l1.Head \wedge e \neq GUARD \wedge e \neq u\}$
```
27  z := y.next;
```
$\{\exists c.\ \mathsf{adjacent\_locked}_{\mathsf{cid}}(None, l1.\mathsf{Head}, S_1, x, GUARD, y, e, z) * \mathsf{adjacent\_notin}_{\mathsf{cid}}(None, l2.\mathsf{Head}, S_2, m, c, n, u, \mathtt{null}, e)$
$* \mathsf{p2}_{\mathsf{cid}}(true, 1, 2, 0, MOVE) \wedge x = l1.Head \wedge e \neq GUARD \wedge e \neq u\}$
```
28  <x.next := z;  todel := Some(l1,e);>
```
$\{\exists c.\ \mathsf{adjacent}_{\mathsf{cid}}(todel, l1.\mathsf{Head}, S_1, x, GUARD, z, \_, \_) * \mathsf{adjacent\_notin}_{\mathsf{cid}}(todel, l2.\mathsf{Head}, S_2, m, c, n, u, \mathtt{null}, e)$
$* \mathsf{L}_{\mathsf{cid}}(y, e, z) * \mathsf{p2}_{\mathsf{cid}}(true, 1, 2, 0, MOVE) \wedge x = l1.Head \wedge e \neq GUARD \wedge e \neq u \wedge todel = Some(l1, e)\}$
```
29  y.next := m.next;
```
$\{\exists c.\ \mathsf{adjacent}_{\mathsf{cid}}(todel, l1.\mathsf{Head}, S_1, x, GUARD, z, \_, \_) * \mathsf{adjacent\_notin}_{\mathsf{cid}}(todel, l2.\mathsf{Head}, S_2, m, c, n, u, \mathtt{null}, e)$
$* \mathsf{L}_{\mathsf{cid}}(y, e, n) * \mathsf{p2}_{\mathsf{cid}}(true, 1, 2, 0, MOVE) \wedge x = l1.Head \wedge e \neq GUARD \wedge e \neq u \wedge todel = Some(l1, e)\}$
```
30  <m.next := y;  toadd_cid := false; todel := None;>
```
$\{\exists c.\ \mathsf{adjacent}_{\mathsf{cid}}(None, l1.\mathsf{Head}, S_1, x, GUARD, z, \_, \_) * \mathsf{adjacent}_{\mathsf{cid}}(None, l2.\mathsf{Head}, S_2, m, c, y, e, n)$
$* \mathsf{p2}_{\mathsf{cid}}(false, 1, 2, 0, skip)\}$
```
31  unlock(x);
32  unlock(y);
33  unlock(m);
34  <l2.level := 1;>
35  unlock(mutex);
```
$\{\exists todel.\ \mathsf{ls\_irr}_{\mathsf{cid}}(todel, l1.\mathsf{Head}, S_1) * \mathsf{ls\_irr}_{\mathsf{cid}}(todel, l2.\mathsf{Head}, S_2) * \mathsf{p1}_{\mathsf{cid}}(false, \_, \_, 0, skip)\}$
```
36  return true;
```

Fig. 68. Proof outline of move method (2).