Modular Verification of Linearizability with Non-Fixed Linearization Points (Extended Version)

Hongjin Liang Xinyu Feng
University of Science and Technology of China
lhj1018@mail.ustc.edu.cn xyfeng@ustc.edu.cn

Abstract

Locating linearization points (LPs) is an intuitive approach for proving linearizability, but it is difficult to apply the idea in Hoare-style logic for formal program verification, especially for verifying algorithms whose LPs cannot be statically located in the code. In this paper, we propose a program logic with a lightweight instrumentation mechanism which can verify algorithms with non-fixed LPs, including the most challenging ones that use the helping mechanism to achieve lock-freedom (as in HSY elimination-based stack), or have LPs depending on unpredictable future executions (as in the lazy set algorithm), or involve both features. We also develop a thread-local simulation as the meta-theory of our logic, and show it implies contextual refinement, which is equivalent to linearizability. Using our logic we have successfully verified various classic algorithms, some of which are used in the `java.util.concurrent` package.

1. Introduction

Linearizability is a standard correctness criterion for concurrent object implementations [16]. It requires the fine-grained implementation of an object operation to have the same effect with an instantaneous atomic operation. To prove linearizability, the most intuitive approach is to find a linearization point (LP) in the code of the implementation, and show that it is the single point where the effect of the operation takes place.

However, it is difficult to apply this idea when the LPs are not fixed in the code of object methods. For a large class of lock-free algorithms with *helping* mechanism (*e.g.*, HSY elimination-based stack [14]), the LP of one method might be in the code of some other method. In these algorithms, each thread maintains a descriptor recording all the information required to fulfill its intended operation. When a thread A detects conflicts with another thread B, A may access B's descriptor and help B finish its intended operation first before finishing its own. In this case, B's operation takes effect at a step from A. Thus its LP should *not* be in its own code, but in the code of thread A.

Besides, in optimistic algorithms and lazy algorithms (*e.g.*, Heller *et al.*'s lazy set [13]), the LPs might depend on unpredictable future interleavings. In those algorithms, a thread may access the shared states as if no interference would occur, and validate the accesses later. If the validation succeeds, it finishes the operation; otherwise it rolls back and retries. Its LP is usually at a prior state access, but only if the later validation succeeds.

Reasoning about algorithms with non-fixed LPs has been a long-standing problem. Most existing work either supports only simple objects with static LPs in the implementation code (*e.g.*, [2, 5, 18, 29]), or lacks formal soundness arguments (*e.g.*, [31]). In this paper, we propose a program logic for verification of linearizability with non-fixed LPs. For a concrete implementation of an object method, we treat the corresponding abstract atomic operation and the abstract state as auxiliary states, and insert auxiliary commands at the LP to execute the abstract operation simultaneously with the concrete step. We verify the instrumented implementation in an existing concurrent program logic (we will use LRG [8] in this paper), but extend it with new logic rules for the auxiliary commands. We also give a new relational interpretation to the logic assertions, and show that at the LP, the step of the original concrete implementation has the same effect as the abstract operation. We handle non-fixed LPs in the following way:

- To support the helping mechanism, we collect a pending thread pool as auxiliary state, which is a set of threads and their abstract operations that might be helped. We allow the thread that is currently being verified to use auxiliary commands to help execute the abstract operations in the pending thread pool.
- For future-dependent LPs, we introduce a try-commit mechanism to reason with uncertainty. The **try** clause guesses whether the corresponding abstract operation should be executed and keeps all possibilities, while **commit** chooses a specific possible case when we know which guess is correct later.

Although our program logic looks intuitive, it is challenging to prove that the logic is sound *w.r.t.* linearizability. Recent work has shown the equivalence between linearizability and contextual refinement [5, 9, 10]. The latter is often verified by proving simulations between the concrete implementation and the atomic operation [5]. The simulation establishes some correspondence between the executions of the two sides, showing there exists one step in the concrete execution that fulfills the abstract operation. Given the equivalence between linearizability and refinement, we would expect the simulations to justify the soundness of the LP method and to serve as the meta-theory of our logic. However, existing thread-local simulations do not support non-fixed LPs (except the recent work [30], which we will discuss in Sec. 7). We will explain the challenges in detail in Sec. 2.

Our work is inspired by the earlier work on linearizability verification, in particular the use of auxiliary code and states by Vafeiadis [31] and our previous work on thread-local simulation RGSim [18], but makes the following new contributions:

- We propose the first program logic that has a formal soundness proof for linearizability with non-fixed LPs. Our logic is built upon the unary program logic LRG [8], but we give a relational interpretation of assertions and rely/guarantee conditions. We also introduce new logic rules for auxiliary commands used specifically for linearizability proofs.

- We give a light instrumentation mechanism to relate concrete implementations with abstract operations. The systematic use of auxiliary states and commands makes it possible to execute the abstract operations synchronously with the concrete code. The try-commit clauses allow us to reason about future-dependent uncertainty without resorting to prophecy variables [1, 31], whose existing semantics (*e.g.*, [1]) is unsuitable for Hoare-style verification.
- We design a novel thread-local simulation as the meta-theory for our logic. It generalizes RGSim [18] and other compositional reasoning of refinement (*e.g.*, [5, 29]) with the support for non-fixed LPs.
- Instead of ensuring linearizability directly, the program logic and the simulation both establish contextual refinement, which we prove is equivalent to linearizability. A program logic for contextual refinement is interesting in its own right, since contextual refinement is also a widely accepted (and probably more natural) correctness criterion for library code.
- We successfully apply our logic to verify 12 well-known algorithms. Some of them are used in the `java.util.concurrent` package, such as MS non-blocking queue [22] and Harris-Michael lock-free list [11, 21].

In the rest of this paper, we first analyze the challenges in the logic design and explain our approach informally in Sec. 2. Then we give the basic technical setting in Sec. 3, including a formal operational definition of linearizability. We present our program logic in Sec. 4, and the new simulation relation as the meta-theory in Sec. 5. In Sec. 6 we summarize all the algorithms we have verified and sketch the proofs of three representative algorithms. We discuss related work and conclude in Sec. 7.

2. Challenges and Our Approach

Below we start from a simple program logic for linearizability with fixed LPs, and extend it to support algorithms with non-fixed LPs. We also discuss the problems with the underlying meta-theory, which establishes the soundness of the logic *w.r.t.* linearizability.

2.1 Basic Logic for Fixed LPs

We first show a simple and intuitive logic which follows the LP approach. As a working example, Fig. 1(a) shows the implementation of push in Treiber stack [28] (let's first ignore the blue code at line `7'`). The stack object is implemented as a linked list pointed to by `S`, and `push(v)` repeatedly tries to update `S` to point to the new node using compare-and-swap (`cas`) until it succeeds.

To verify linearizability, we first locate the LP in the code. The LP of `push(v)` is at the `cas` statement when it succeeds (line `7`). That is, the successful `cas` can correspond to the abstract atomic `PUSH(v)` operation: `Stk := v::Stk`; and all the other concrete steps cannot. Here we simply represent the abstract stack `Stk` as a sequence of values with "`::`" for concatenation. Then `push(v)` can be linearized at the successful `cas` since it is the single point where the operation takes effect.

We can encode the above reasoning in an existing (unary) concurrent program logic, such as Rely-Guarantee reasoning [17] and CSL [23]. Inspired by Vafeiadis [31], we embed the abstract operation γ and the abstract state θ as auxiliary states on the concrete side, so the program state now becomes $(\sigma, (\gamma, \theta))$, where σ is the original concrete state. Then we instrument the concrete implementation with an auxiliary command **linself** (shorthand for "linearize self") at the LP to update the auxiliary state. Intuitively, **linself** will execute the abstract operation γ over the abstract state θ, as described in the following operational semantics rule:

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

(a) Treiber Stack

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

(c) Pair Snapshot

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

(b) HSY Elimination-Based Stack

Figure 1. LPs and Instrumented Auxiliary Commands

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

Here $\rightsquigarrow$ encodes the transition of γ at the abstract level, and **end** is a termination marker. We insert **linself** into the same atomic block with the concrete statement at the LP, such as line `7'` in Fig. 1(a), so that the concrete and abstract sides are executed simultaneously. Here the atomic block $\langle C \rangle$ means C is executed atomically. Then we reason about the instrumented code using a traditional concurrent logic extended with a new inference rule for **linself**.

The idea is intuitive, but it cannot handle more advanced algorithms with non-fixed LPs, including the algorithms with the helping mechanism and those whose locations of LPs depend on the future interleavings. Below we analyze the two challenges in detail and explain our solutions using two representative algorithms, the HSY stack and the pair snapshot.

2.2 Support Helping Mechanism with Pending Thread Pool

HSY elimination-based stack [14] is a typical example using the helping mechanism. Figure 1(b) shows part of its push method implementation. The basic idea behind the algorithm is to let a push and a pop cancel out each other.

At the beginning of the method in Fig. 1(b), the thread allocates its *thread descriptor* (line `3`), which contains the thread id, the name of the operation to be performed, and the argument. The current thread `cid` first tries to perform Treiber stack's push (line `5`). It returns if succeeds. Otherwise, it writes its descriptor in the global `loc` array (line `6`) to allow other threads to eliminate its push. The elimination array `loc`$[1..n]$ has one slot for each thread, which holds the pointer to a thread descriptor. The thread randomly reads a slot `him` in `loc` (line `7`). If the descriptor `q` says `him` is doing pop, `cid` tries to eliminate itself with `him` by two `cas` instructions. The first clears `cid`'s entry in `loc` so that no other thread could eliminate with `cid` (line `9`). The second attempts to mark the entry of `him` in `loc` as "eliminated with `cid`" (line `10`). If successful, it should be the LPs of *both* the push of `cid` and the pop of `him`, with the push happening immediately before the pop.

The helping mechanism allows the current thread to linearize the operations of other threads, which cannot be expressed in the basic logic. It also breaks modularity and makes thread-local verification difficult. For the thread `cid`, its concrete step could correspond to the steps of both `cid` and `him` at the abstract level. For `him`, a step from its environment could fulfill its abstract operation. We must ensure in the thread-local verification that the two threads `cid` and `him` always take consistent views on whether and how the abstract operation of `him` is done. For example, if we let a concrete step in `cid` fulfill the abstract pop of `him`, we must know `him` is indeed doing pop and its pop has not been done before. Otherwise, we will not be able to compose `cid` and `him` in parallel.

We extend the basic logic to express the helping mechanism. First we introduce a new auxiliary command **lin**(t) to linearize a specific thread `t`. For instance, in Fig. 1(b) we insert line `10'` at the LP to execute both the push of `cid` and the pop of `him` at the abstract level. We also extend the auxiliary state to record both abstract operations of `cid` and `him`. More generally, we embed a pending thread pool U, which maps threads to their abstract operations. It specifies a set of threads whose operations might be helped by others. Then under the new state $(\sigma, (U, \theta))$, the semantics of **lin**(t) just executes the thread `t`'s abstract operation in U, similarly to the semantics of **linself** discussed before.

The shared pending thread pool U allows us to recover the thread modularity when verifying the helping mechanism. A concrete step of `cid` could fulfill the operation of `him` in U as well as its own abstract operation; and conversely, the thread `him` running in parallel could check U to know if its operation has been finished by others (such as `cid`) or not. We gain consistent abstract information of other threads in the thread-local verification. Note that the need of U itself does not break modularity because the required information of other threads' abstract operations can be inferred from the concrete state. In the HSY stack example, we know `him` is doing pop by looking at its thread descriptor in the elimination array. In this case U can be viewed as an abstract representation of the elimination array.

2.3 Try-Commit Commands for Future-Dependent LPs

Another challenge is to reason about optimistic algorithms whose LPs depend on the future interleavings.

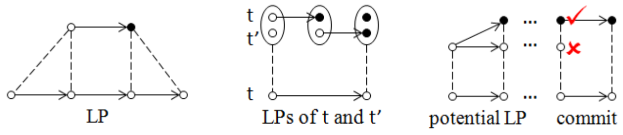
We give a toy example, pair snapshot [26], in Fig. 1(c). The object is an array `m`, each slot of which contains two fields: `d` for the data and `v` for the version number. The `write(i,d)` method (lines 9) updates the data stored at address `i` and increments the version number instantaneously. The `readPair(i,j)` method intends to perform an atomic read of two slots `i` and `j` in the presence of concurrent writes. It reads the data at slots `i` and `j` separately at lines 4 and 5, and validate the first read at line 6. If `i`'s version number has not been increased, the thread knows that when it read `j`'s data at line 5, `i`'s data had not been updated. This means the two reads were at a consistent state, thus the thread can return. We can see that the LP of `readPair` should be at line 5 when the thread reads `j`'s data, but only if the validation at line 6 succeeds. That is, whether we should linearize the operation at line 5 depends on the future unpredictable behavior of line 6.

As discussed a lot in previous work (*e.g.*, [1, 31]), the future-dependent LPs cannot be handled by introducing history variables, which are auxiliary variables storing values or events in the past executions. We have to refer to events coming from the unpredictable future. Thus people propose prophecy variables [1, 31] as the dual of history variables to store future behaviors. But as far as we know, there is no semantics of prophecy variables suitable for Hoare-style local and compositional reasoning.

Instead of resorting to prophecy variables, we follow the speculation idea [30]. For the concrete step at a potential LP (*e.g.*, line 5 of `readPair`), we execute the abstract operation speculatively and keep both the result and the original abstract configuration. Later based on the result of the validation (*e.g.*, line 6 in `readPair`), we keep the appropriate branch and discard the other.

For the logic, we introduce two new auxiliary commands: **trylinself** is to do speculation, and **commit**(p) will commit to the appropriate branch satisfying the assertion p. In Fig. 1(c), we insert lines `5'` and `6'`, where $\texttt{cid} \rightarrowtail (\textbf{end}, (\texttt{a}, \texttt{b}))$ means that the current thread `cid` should have done its abstract operation and would return $(\texttt{a}, \texttt{b})$. We also extend the auxiliary state to record the multiple possibilities of abstract operations and abstract states after speculation.

Furthermore, we can combine the speculation idea with the pending thread pool. We allow the abstract operations in the pending thread pool as well as the current thread to speculate. Then we could handle some trickier algorithms such as RDCSS [12], in which the location of LP for thread `t` may be in the code of some other thread and also depend on the future behaviors of that thread. Please see Sec. 6 for one such example.



(a) Simple Simulation (b) Pending Thread Pool (c) Speculation

Figure 2. Simulation Diagrams

2.4 Simulation as Meta-Theory

The LP proof method can be understood as building simulations between the concrete implementations and the abstract atomic operations, such as the simple weak simulation in Fig. 2(a). The lower-level and higher-level arrows are the steps of the implementation and of the abstract operation respectively, and the dashed lines denote the simulation relation. We use dark nodes and white nodes at the abstract level to distinguish whether the operation has been finished or not. The only step at the concrete side corresponding to the single abstract step should be the LP of the implementation (labeled "LP" in the diagram). Since our program logic is based on the LP method, we can expect simulations to justify its soundness. In particular, we want a *thread-local* simulation which can handle both the helping mechanism and future-dependent LPs and can ensure linearizability.

To support helping in the simulation, we should allow the LP step at the concrete level to correspond to an abstract step made by a thread other than the one being verified. This requires information from other threads at the abstract side, thus makes it difficult to build a thread-local simulation. To address the problem, we introduce the pending thread pool at the abstract level of the simulation, just as in the development of our logic in Sec. 2.2. The new simulation is shown in Fig. 2(b). We can see that a concrete step of thread `t` could help linearize the operation of `t'` in the pending thread pool as well as its own operation. Thus the new simulation intuitively supports the helping mechanism.

As forward simulations, neither of the simulations in Fig. 2(a) and (b) supports future-dependent LPs. For each step along the concrete execution in those simulations, we need to decide immediately whether the step is at the LP, and cannot postpone the decision to the future. As discussed a lot in previous work (*e.g.*, [1, 3, 6, 20]), we have to introduce backward simulations or hybrid simulations to support future-dependent LPs. Here we exploit the speculation idea and develop a forward-backward simulation [20]. As shown in Fig. 2(c), we keep both speculations after the potential LP, where

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

Figure 3. Syntax of the Programming Language

$$
\begin{array}{rcll}
(\textit{ThrdID}) & \mathtt{t} & \in & \textit{Nat} \\
(\textit{Mem}) & \sigma & \in & (\textit{PVar} \cup \textit{Nat}) \rightharpoonup \textit{Int} \\
(\textit{CallStk}) & \kappa & ::= & (\sigma_l, x, C) \mid \circ \\
(\textit{ThrdPool}) & \mathcal{K} & ::= & \{\mathtt{t}_1 \rightsquigarrow \kappa_1,\ldots,\mathtt{t}_n \rightsquigarrow \kappa_n\} \\
(\textit{PState}) & \mathcal{S} & ::= & (\sigma_c,\sigma_o,\mathcal{K}) \\
(\textit{LState}) & s & ::= & (\sigma_c,\sigma_o,\kappa) \\
(\textit{Evt}) & e & ::= & (\mathtt{t},f,n) \mid (\mathtt{t},\textbf{ok},n) \mid (\mathtt{t},\textbf{obj},\textbf{abort}) \\
 & & & \mid (\mathtt{t},\textbf{out},n) \mid (\mathtt{t},\textbf{clt},\textbf{abort}) \\
(\textit{ETrace}) & H & ::= & \epsilon \mid e :: H
\end{array}
$$

Figure 4. States and Event Traces

the higher black nodes result from executing the abstract operation and the lower white nodes record the original abstract configuration. Then at the validation step we commit to the correct branch.

Finally, to ensure linearizability, the thread-local simulation has to be *compositional*. As a counterexample, we can construct a simple simulation (like the one in Fig. 2(a)) between the following implementation C and the abstract atomic increment operation γ, but C is not linearizable *w.r.t.* γ.

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

The reason is that the simple simulation is not compositional *w.r.t.* parallel compositions. To address this problem, we proposed a compositional simulation RGSim [18] in previous work. The idea is to parameterize the simple simulation with the interference with the environment, in the form of rely/guarantee conditions (R and G) [17]. RGSim says, the concrete executions are simulated by the abstract executions under interference from the environment R, and all the related state transitions of the thread being verified should satisfy G. For parallel composition, we check that the guarantee G of each thread is permitted in the rely R of the other. Then the simulation becomes compositional and can ensure linearizability.

We combine the above ideas and develop a new compositional simulation with the support of non-fixed LPs as the meta-theory of our logic. We will discuss our simulation formally in Sec. 5.

3. Basic Technical Settings and Linearizability

In this section, we formalize linearizability of an object implementation *w.r.t.* its specification, and show that linearizability is equivalent to contextual refinement.

3.1 Language and Semantics

As shown in Fig. 3, a program W contains several client threads in parallel, each of which could call the methods declared in the object Π. A method is defined as a pair (x, C), where x is the formal argument and C is the method body. For simplicity, we assume there is only one object in W and each method takes one argument only, but it is easy to extend our work with multiple objects and arguments.

Each method returns a value to the client using the **return** E command, unless it does not terminate. We use a runtime command **noret** to abort methods that terminate but do not execute **return** E. It is automatically appended to the method code and is not supposed to be used by programmers. Other commands are mostly standard. Commands $x := [E]$ and $[E] := E'$ do memory load and store. Memory allocation and free are done by $x := \textbf{cons}(E_1, \ldots, E_n)$ and $\textbf{dispose}(E)$. The atomic block $\langle C \rangle$ executes C atomically. Clients can also use $\textbf{print}(E)$ to produce observable external events. We do not allow the object's methods to produce external events. To simplify the semantics, we also assume there are no nested method calls.

Figure 4 gives the model of program states. Memory σ maps variables and memory locations to integers. To ensure that clients can access the object via calling its methods only, we first need to precisely determine the object data from a whole state. Here we partition a global state $\mathcal{S}$ into the client memory σ_c, the object σ_o and a thread pool $\mathcal{K}$. The thread pool maps thread identifiers $\mathtt{t}$ to their local call stack frames. A call stack κ could be either empty ($\circ$) when the thread is not executing a method, or a triple (σ_l, x, C), where σ_l maps the method's formal argument and local variables (if any) to their values, x is the caller's variable to receive the return value, and C is the caller's remaining code to be executed after the method returns. To give a thread-local semantics, we also define the thread local view s of the state.

Figure 5 gives selected rules of the operational semantics. We show three kinds of transitions: $\longmapsto$ for the top-level program transitions, $\longrightarrow_{\mathtt{t},\Pi}$ for the transitions of thread $\mathtt{t}$ with the methods' declaration Π, and $\longrightarrow\!\!\!\!\triangleright_{\mathtt{t}}$ for the steps inside method calls of thread $\mathtt{t}$. To describe the operational semantics for threads, we use an execution context $\mathbf{E}$:

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

The hole $[\,]$ shows the place where the execution of code occurs. $\mathbf{E}[\,C\,]$ represents the code resulting from placing C into the hole.

We label transitions with events e defined in Fig. 4. An event could be a method invocation $(\mathtt{t}, f, n)$ or return $(\mathtt{t}, \textbf{ok}, n)$, a fault $(\mathtt{t}, \textbf{obj}, \textbf{abort})$ produced by the object method code, an output $(\mathtt{t}, \textbf{out}, n)$ generated by $\textbf{print}(E)$, or a fault $(\mathtt{t}, \textbf{clt}, \textbf{abort})$ from the client code. The first two events are called object events, and the last two are observable external events. The third one $(\mathtt{t}, \textbf{obj}, \textbf{abort})$ belongs to both classes. Note that here we explicitly distinguish the faults caused by the methods and by the clients. This allows us to clearly know where to place the blame when the program aborts, and then to discuss the safety of the object. An event trace H is then defined as a finite sequence of events.

The operational semantics is mostly straightforward. Note that **noret** is appended at the end of the method body at the time of method invocation. Since **noret** aborts the program, a safe method implementation must end with **return** E.

3.2 Object Specification and Linearizability

Next we formalize object specifications Γ, which maps method names to their abstract operations γ, as shown in Fig. 6. γ transforms an argument value and an initial abstract object to a return value with a resulting abstract object in a single step. It specifies the intended sequential behaviors of the method, which should be always safe. Here we model γ as a partial function, thus it could be blocked at certain abstract object θ (when $\theta \notin \textit{dom}(\gamma(n))$ for some n). For example, when a thread attempts to dequeue from an empty queue, it may need to wait until another thread enqueues an item. The abstract object representation θ is defined as a mapping from program variables to abstract values. We leave the abstract values unspecified here, which can be instantiated by programmers.

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

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

(a) Program Transitions

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

$$\frac{\kappa = (\sigma_l, x, C) \quad [\![E]\!]_{\sigma_o \uplus \sigma_l} = n \quad \sigma_c' = \sigma_c\{x \leadsto n\}}{(\mathbf{E}[\,\textbf{return}\ E\,], (\sigma_c, \sigma_o, \kappa)) \xrightarrow{(\mathsf{t}, \textbf{ok}, n)}_{\mathsf{t},\Pi} (C, (\sigma_c', \sigma_o, \circ))} \qquad \frac{\kappa = (\sigma_l, x, C) \quad [\![E]\!]_{\sigma_o \uplus \sigma_l}\ \mathit{undefined}}{(\mathbf{E}[\,\textbf{return}\ E\,], (\sigma_c, \sigma_o, \kappa)) \xrightarrow{(\mathsf{t}, \textbf{obj}, \textbf{abort})}_{\mathsf{t},\Pi} \textbf{abort}}$$

$$\frac{}{(\textbf{noret}, s) \xrightarrow{(\mathsf{t}, \textbf{obj}, \textbf{abort})}_{\mathsf{t},\Pi} \textbf{abort}} \qquad \frac{[\![E]\!]_{\sigma_c} = n}{(\mathbf{E}[\,\textbf{print}(E)\,], (\sigma_c, \sigma_o, \circ)) \xrightarrow{(\mathsf{t}, \textbf{out}, n)}_{\mathsf{t},\Pi} (\mathbf{E}[\,\textbf{skip}\,], (\sigma_c, \sigma_o, \circ))}$$

$$\frac{(C, \sigma_o \uplus \sigma_l) \rightarrowtriangle_{\mathsf{t}} (C', \sigma_o' \uplus \sigma_l') \quad \mathit{dom}(\sigma_l) = \mathit{dom}(\sigma_l')}{(C, (\sigma_c, \sigma_o, (\sigma_l, x, C_c))) \longrightarrow_{\mathsf{t},\Pi} (C', (\sigma_c, \sigma_o', (\sigma_l', x, C_c)))} \qquad \frac{(C, \sigma_o \uplus \sigma_l) \rightarrowtriangle_{\mathsf{t}} \textbf{abort}}{(C, (\sigma_c, \sigma_o, (\sigma_l, x, C_c))) \xrightarrow{(\mathsf{t}, \textbf{obj}, \textbf{abort})}_{\mathsf{t},\Pi} \textbf{abort}}$$

(b) Thread Transitions

$$\frac{\{l, \ldots, l{+}i{-}1\} \cap \mathit{dom}(\sigma) = \emptyset \quad [\![E_1]\!]_\sigma = n_1 \quad \ldots \quad [\![E_i]\!]_\sigma = n_i \quad x \in \mathit{dom}(\sigma)}{(\mathbf{E}[\,x := \textbf{cons}(E_1, \ldots, E_i)\,], \sigma) \rightarrowtriangle_{\mathsf{t}} (\mathbf{E}[\,\textbf{skip}\,], (\sigma\{x \leadsto l\}) \uplus \{l \leadsto n_1, \ldots, l{+}i{-}1 \leadsto n_i\})}$$

$$\frac{[\![E_j]\!]_\sigma\ \mathit{undefined} \quad (1 \le j \le i) \quad \text{or} \quad x \notin \mathit{dom}(\sigma)}{(\mathbf{E}[\,x := \textbf{cons}(E_1, \ldots, E_i)\,], \sigma) \rightarrowtriangle_{\mathsf{t}} \textbf{abort}} \qquad \frac{(C, \sigma) \rightarrowtriangle_{\mathsf{t}}^{*} (\textbf{skip}, \sigma')}{(\mathbf{E}[\,\langle C \rangle\,], \sigma) \rightarrowtriangle_{\mathsf{t}} (\mathbf{E}[\,\textbf{skip}\,], \sigma')} \qquad \frac{(C, \sigma) \rightarrowtriangle_{\mathsf{t}}^{*} \textbf{abort}}{(\mathbf{E}[\,\langle C \rangle\,], \sigma) \rightarrowtriangle_{\mathsf{t}} \textbf{abort}}$$

(c) Thread Transitions Inside Method Calls

Figure 5. Selected Rules of Concrete Operational Semantics

(*AbsObj*)	θ	$\in$	$\mathit{PVar} \rightharpoonup \mathit{AbsVal}$
(*MSpec*)	γ	$\in$	$\mathit{Int} \to \mathit{AbsObj} \rightharpoonup \mathit{Int} \times \mathit{AbsObj}$
(*OSpec*)	Γ	$::=$	$\{f_1 \leadsto \gamma_1, \ldots, f_n \leadsto \gamma_n\}$
(*AbsStmt*)	$\mathbb{C}$	$::=$	$C \mid \textbf{fexec}(f, n) \mid \textbf{fret}(n)$
(*AbsProg*)	$\mathbb{W}$	$::=$	$\textbf{skip} \mid \textbf{with}\ \Gamma\ \textbf{do}\ \mathbb{C} \parallel \ldots \parallel \mathbb{C}$
(*AbsStk*)	*ak*	$::=$	$(x, C) \mid \circ$
(*AbsPool*)	$\mathbb{K}$	$::=$	$\{\mathsf{t}_1 \leadsto \mathit{ak}_1, \ldots, \mathsf{t}_n \leadsto \mathit{ak}_n\}$
(*AbsState*)	*as*	$::=$	$(\sigma_c, \theta, \mathit{ak})$
(*AbsPState*)	$\mathbb{S}$	$::=$	$(\sigma_c, \theta, \mathbb{K})$

Figure 6. Object Specification and Abstract Machine

Then we give an abstract version of programs, where clients interact with the abstract object specification. Behaviors of this abstract program captures the intended behaviors of the original program where concrete object implementation is used. Syntax of the language is also defined in Fig. 6. Here in the program $\mathbb{W}$ client threads use the object specification Γ instead of its implementation Π. Statements $\mathbb{C}$ for client threads consist of all statements C in the concrete language and two extra runtime commands **fexec** and **fret** used in the intermediate steps at the method calls, which will be discussed later. An abstract state $\mathbb{S}$ consists of the client memory σ_c, the abstract object θ and the abstract thread pool $\mathbb{K}$. Here in the abstract stack frame *ak* we do not need the local memory for the method execution as in κ. The thread local view of states are now represented as *as*.

Selected semantic rules for the abstract programs is shown in Fig. 7, which is similar to the concrete semantics. Below we only discuss the rules for method calls. Although the method specification γ is atomic, we split the method call $x := f(E)$ into three steps to decouple the evaluation of the argument E and the assignment of the return value to x from the execution of the atomic γ. When a client thread calls a method f, it first computes the argument value n and saves the client information in the call stack *ak*, as in the concrete semantics. This step reduces to $\textbf{fexec}(f, n)$, and is the preparation phase of the method call. The second step executes $\textbf{fexec}(f, n)$ to $\textbf{fret}(n')$ respecting the specification of f, and updates the abstract object atomically. A pair of invocation and return events is generated during this step. Finally $\textbf{fret}(n')$ finishes the method call and resumes the client executions.

Linearizability Linearizability [16] is defined using the notion of histories, which are special event traces H consisting of only object events (*i.e.*, invocations, returns and object faults).

Below we use $H(i)$ for the i-th event of H, and $|H|$ for the length of H. $H|_{\mathsf{t}}$ represents the sub-history consisting of all the events whose thread id is t. The predicates $\mathsf{is\_inv}(e)$ and $\mathsf{is\_res}(e)$ mean that the event e is a method invocation and a response (*i.e.*, a return or an object fault) respectively.

- $\mathsf{is\_inv}(e)$ iff there exist t, f and n such that $e = (\mathsf{t}, f, n)$;
- $\mathsf{is\_ok}(e)$ iff there exist t and n' such that $e = (\mathsf{t}, \textbf{ok}, n')$;
- $\mathsf{is\_abt}(e)$ iff there exists t such that $e = (\mathsf{t}, \textbf{obj}, \textbf{abort})$;
- $\mathsf{is\_res}(e)$ iff either $\mathsf{is\_ok}(e)$ or $\mathsf{is\_abt}(e)$ holds.

$$\frac{(\mathbb{C}_i, (\sigma_c, \theta, \mathbb{K}(i))) \circ\!\!\xrightarrow{H}_{i,\Gamma} (\mathbb{C}'_i, (\sigma'_c, \theta', ak'))}{(\textbf{with}\ \Gamma\ \textbf{do}\ \mathbb{C}_1 \,\|\, \ldots \mathbb{C}_i \ldots \,\|\, \mathbb{C}_n, (\sigma_c, \theta, \mathbb{K})) \ \phi\!\!\xrightarrow{H} (\textbf{with}\ \Gamma\ \textbf{do}\ \mathbb{C}_1 \,\|\, \ldots \mathbb{C}'_i \ldots \,\|\, \mathbb{C}_n, (\sigma'_c, \theta', \mathbb{K}\{i \leadsto ak'\}))}$$

$$\frac{}{(\textbf{with}\ \Gamma\ \textbf{do}\ \textbf{skip} \,\|\, \ldots \,\|\, \textbf{skip}, \mathbb{S}) \ \phi\!\!\longrightarrow (\textbf{skip}, \mathbb{S})} \qquad \frac{(\mathbb{C}_i, (\sigma_c, \theta, \mathbb{K}(i))) \circ\!\!\xrightarrow{H}_{i,\Gamma} \textbf{abort}}{(\textbf{with}\ \Gamma\ \textbf{do}\ \mathbb{C}_1 \,\|\, \ldots \mathbb{C}_i \ldots \,\|\, \mathbb{C}_n, (\sigma_c, \theta, \mathbb{K})) \ \phi\!\!\xrightarrow{H} \textbf{abort}}$$

$$\frac{f \in \mathit{dom}(\Gamma) \qquad [\![E]\!]_{\sigma_c} = n \qquad x \in \mathit{dom}(\sigma_c) \qquad ak = (x, \mathbf{E}[\,\textbf{skip}\,])}{(\mathbf{E}[\,x := f(E)\,], (\sigma_c, \theta, \circ)) \circ\!\!\longrightarrow_{\mathsf{t},\Gamma} (\textbf{fexec}(f, n), (\sigma_c, \theta, ak))} \qquad \frac{\Gamma(f)(n)(\theta) = (n', \theta')}{(\textbf{fexec}(f, n), (\sigma_c, \theta, ak)) \circ\!\!\xrightarrow{(\mathsf{t}, f, n)::(\mathsf{t}, \textbf{ok}, n')}_{\mathsf{t},\Gamma} (\textbf{fret}(n'), (\sigma_c, \theta', ak))}$$

$$\frac{ak = (x, C) \qquad \sigma'_c = \sigma_c\{x \leadsto n'\}}{(\textbf{fret}(n'), (\sigma_c, \theta, ak)) \circ\!\!\longrightarrow_{\mathsf{t},\Gamma} (C, (\sigma'_c, \theta, \circ))} \qquad \frac{f \notin \mathit{dom}(\Gamma) \quad \text{or} \quad [\![E]\!]_{\sigma_c}\ \textit{undefined} \quad \text{or} \quad x \notin \mathit{dom}(\sigma_c)}{(\mathbf{E}[\,x := f(E)\,], (\sigma_c, \theta, \circ)) \circ\!\!\xrightarrow{(\mathsf{t}, \textbf{clt}, \textbf{abort})}_{\mathsf{t},\Gamma} \textbf{abort}}$$

Figure 7. Selected Rules of the Abstract Operational Semantics

We say a responses e_2 *matches* an invocation e_1 iff they have the same thread IDs.

$$\begin{aligned}\mathsf{match}(e_1, e_2) \ &\overset{\text{def}}{=}\ \mathsf{is\_inv}(e_1) \wedge \mathsf{is\_res}(e_2)\\ &\quad \wedge (\mathsf{get\_thrd}(e_1) = \mathsf{get\_thrd}(e_2))\end{aligned}$$

A history H is *sequential* iff the first event of H is an invocation, and each invocation, except possibly the last, is immediately followed by a matching response. It is inductively defined as follows.

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

Then H is *well-formed* iff every thread sub-history $H|_{\mathsf{t}}$ is sequential.

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

H is *complete* iff it is well-formed and every invocation has a matching response. An invocation is *pending* if no matching response follows it. We handle pending invocations in an incomplete history H following the standard linearizability definition [16]: we append zero or more response events to H, and drop the remaining pending invocations. Then we get a set of complete histories, which is denoted by $\mathsf{completions}(H)$. Formally, we define $\mathsf{completions}(H)$ as follows.

Definition 1 (Extensions of a history). $\mathsf{extensions}(H)$ is a set of well-formed histories where we extend H by appending successful return events:

$$\frac{\mathsf{well\_formed}(H)}{H \in \mathsf{extensions}(H)}$$

$$\frac{H' \in \mathsf{extensions}(H) \qquad \mathsf{is\_ok}(e) \qquad \mathsf{well\_formed}(H' :: e)}{H' :: e \in \mathsf{extensions}(H)}$$

Or equivalently,

$$\begin{aligned}&\mathsf{extensions}(H) \ \overset{\text{def}}{=}\\ &\{H' \mid \mathsf{well\_formed}(H') \wedge \exists H_{ok}.\ H' = H :: H_{ok} \wedge \forall i.\ \mathsf{is\_ok}(H_{ok}(i))\}.\end{aligned}$$

Definition 2 (Completions of a history). $\mathsf{truncate}(H)$ is the maximal complete sub-history of H, which is inductively defined by dropping the pending invocations in H:

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

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

Then $\mathsf{completions}(H) \ \overset{\text{def}}{=}\ \{\mathsf{truncate}(H') \mid H' \in \mathsf{extensions}(H)\}$. It's a set of histories without pending invocations.

Then we can formulate the linearizability relation between well-formed histories, which is a core notion used in the linearizability definition of an object.

Definition 3 (Linearizability Relation between Histories). $H \preceq_{\mathsf{lin}} H'$ iff

1. $\forall \mathsf{t}.\ H|_{\mathsf{t}} = H'|_{\mathsf{t}}$;
2. there exists a bijection $\pi : \{1, \ldots, |H|\} \to \{1, \ldots, |H'|\}$ such that $\forall i.\ H(i) = H'(\pi(i))$ and

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

That is, H is linearizable *w.r.t.* H' if the latter is a permutation of the former, preserving the order of events in single threads (the first condition) and the non-overlapping method calls (the second condition).

Informally, an *object* is linearizable iff all its concurrent histories are linearizable. We generate the concurrent histories of an object by all the possible clients that may use the object, according to the concrete operational semantics (Figure 5). Below we define $\mathcal{H}[\![W, (\sigma_c, \sigma_o)]\!]$ to get the set of histories from the executions of W with the initial client memory σ_c, the shared object σ_o, and empty call stacks for all threads:

$$\begin{aligned}\mathcal{H}[\![W, (\sigma_c, \sigma_o)]\!] \ &\overset{\text{def}}{=}\ \{\mathsf{get\_hist}(H) \mid \exists \mathcal{S}, W', \mathcal{S}'.\ \mathcal{S} = \mathsf{init}(\sigma_c, \sigma_o)\\ &\quad \wedge ((W, \mathcal{S}) \xmapsto{H}{}^{*} (W', \mathcal{S}') \vee (W, \mathcal{S}) \xmapsto{H}{}^{*} \textbf{abort})\}\,, \text{ where}\\ \mathcal{S} = \mathsf{init}(\sigma_c, \sigma_o) &\text{ iff } \exists \mathcal{K}.\ \mathcal{S} = (\sigma_c, \sigma_o, \mathcal{K}) \wedge \forall \mathsf{t}.\ \mathcal{K}(\mathsf{t}) = \circ\end{aligned}$$

We use $\_ \xmapsto{H}{}^{*} \_$ for zero or multiple-step program transitions with an event trace H generated. $\mathsf{get\_hist}(H)$ projects H to the sub-trace consisting of object events only. By the concrete operational semantics in Figure 5, we know that every generated history in $\mathcal{H}[\![W, (\sigma_c, \sigma_o)]\!]$ is well-formed.

Similarly we can generate histories by the abstract semantics (Figure 7) for an abstract program $\mathbb{W}$ that uses the specification. Here we overload the notations used at the concrete level.

$$\begin{aligned}\mathcal{H}[\![\mathbb{W}, (\sigma_c, \theta)]\!] \ &\overset{\text{def}}{=}\ \{\mathsf{get\_hist}(H) \mid \exists \mathbb{S}, \mathbb{W}', \mathbb{S}'.\ \mathbb{S} = \mathsf{init}(\sigma_c, \theta)\\ &\quad \wedge ((\mathbb{W}, \mathbb{S}) \ \phi\!\!\xrightarrow{H}{}^{*} (\mathbb{W}', \mathbb{S}') \vee (\mathbb{W}, \mathbb{S}) \ \phi\!\!\xrightarrow{H}{}^{*} \textbf{abort})\}\,, \text{ where}\\ \mathbb{S} = \mathsf{init}(\sigma_c, \theta) &\text{ iff } \exists \mathbb{K}.\ \mathbb{S} = (\sigma_c, \theta, \mathbb{K}) \wedge \forall \mathsf{t}.\ \mathbb{K}(\mathsf{t}) = \circ\end{aligned}$$

We can see that every history in $\mathcal{H}[\![\mathbb{W}, (\sigma_c, \theta)]\!]$ is sequential from the abstract semantics.

Then a *legal* sequential history H is a history generated by any client using the specification Γ with an initial abstract object θ.

$$\begin{array}{l}\Gamma \rhd (\theta, H) \stackrel{\text{def}}{=} \\ \exists n, C_1, \ldots, C_n, \sigma_c.\ H \in \mathcal{H}[\![(\textbf{with}\ \Gamma\ \textbf{do}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \theta)]\!]\end{array}$$

The legal sequential histories will serve as the criteria for the concurrent histories of an object when defining linearizability. Then an *object* is linearizable iff all its completed concurrent histories are linearizable *w.r.t.* some legal sequential histories.

Definition 4 (Linearizability of Objects). The object's implementation Π is linearizable *w.r.t.* its specification Γ under a refinement mapping φ, denoted by $\Pi \preceq_{\varphi} \Gamma$, iff

$$\begin{array}{l}\forall n, C_1, \ldots, C_n, \sigma_c, \sigma_o, \theta, H. \\ H \in \mathcal{H}[\![(\textbf{let}\ \Pi\ \textbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_o)]\!] \wedge (\varphi(\sigma_o) = \theta) \\ \Longrightarrow\ \exists H_c, H'.\ H_c \in \mathsf{completions}(H) \wedge \Gamma \rhd (\theta, H') \wedge H_c \preceq_{\mathsf{lin}} H'\end{array}$$

Here the mapping φ relates concrete objects to abstract ones:

$$(\textit{RefMap})\ \ \varphi \in \textit{Mem} \rightharpoonup \textit{AbsObj}$$

The side condition $\varphi(\sigma_o) = \theta$ in the above definition requires the initial concrete object σ_o to be a well-formed data structure representing a valid object θ.

3.3 Contextual Refinement and Linearizability

Besides linearizability, we have *contextual refinement*, another widely accepted correctness criteria for object code. Below we formulate its definition and prove the two notions are equivalent.

We first generate the observable event traces using our concrete and abstract semantics, as shown below.

$$\begin{array}{rl}\mathcal{O}[\![W, (\sigma_c, \sigma_o)]\!] \stackrel{\text{def}}{=} & \{\mathsf{get\_obsv}(H) \mid \exists \mathcal{S}, W', \mathcal{S}'.\ \mathcal{S} = \mathsf{init}(\sigma_c, \sigma_o) \\ & \wedge ((W, \mathcal{S}) \xrightarrow{H}{}^{*} (W', \mathcal{S}') \vee (W, \mathcal{S}) \xrightarrow{H}{}^{*} \textbf{abort})\} \\ \mathcal{O}[\![\mathbb{W}, (\sigma_c, \theta)]\!] \stackrel{\text{def}}{=} & \{\mathsf{get\_obsv}(H) \mid \exists \mathbb{S}, \mathbb{W}', \mathbb{S}'.\ \mathbb{S} = \mathsf{init}(\sigma_c, \theta) \\ & \wedge ((\mathbb{W}, \mathbb{S}) \overset{H}{\Longrightarrow}{}^{*} (\mathbb{W}', \mathbb{S}') \vee (\mathbb{W}, \mathbb{S}) \overset{H}{\Longrightarrow}{}^{*} \textbf{abort})\}\end{array}$$

where $\mathsf{get\_obsv}(H)$ projects H to the sub-trace consisting of observable events only.

Then contextual refinement $\Pi \sqsubseteq_{\varphi} \Gamma$ says that, for any client context $C_1 \,\|\, \ldots \,\|\, C_n$, the observable event traces it generates when using Π are not more than those generated when using Γ instead.

Definition 5 (Contextual Refinement). $\Pi \sqsubseteq_{\varphi} \Gamma$ iff

$$\begin{array}{l}\forall n, C_1, \ldots, C_n, \sigma_c, \sigma_o, \theta.\ (\varphi(\sigma_o) = \theta) \\ \quad \Longrightarrow\ \mathcal{O}[\![(\textbf{let}\ \Pi\ \textbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_o)]\!] \\ \qquad \subseteq \mathcal{O}[\![(\textbf{with}\ \Gamma\ \textbf{do}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \theta)]\!].\end{array}$$

Following Filipović *et al.* [9], we can prove that linearizability is equivalent to contextual refinement. The proofs are given in Appendix A.

Theorem 6 (Equivalence). $\Pi \preceq_{\varphi} \Gamma \iff \Pi \sqsubseteq_{\varphi} \Gamma$.

The theorem gives us another point of view to understand linearizability. Since linearizability implies contextual refinement, we can soundly replace the object specification by its implementation without generating more observable behaviors for any client. In particular, the safety of a client using the specification will be preserved when using a linearizable implementation instead, since fault events are observable. On the other hand, since contextual refinement also implies linearizability, we can use various proof methods (such as simulations and logical relations) for the former to verify the latter. In the next section, we will define a new simulation which implies contextual refinement and can verify linearizability of objects that might have non-fixed linearization points.

$$\begin{array}{rrcl}(\textit{InsStmt}) & \widetilde{C} & ::= & \textbf{skip} \mid c \mid \textbf{return}\ E \mid \textbf{noret} \\ & & \mid & \textbf{linself} \mid \textbf{lin}(E) \mid \textbf{trylinself} \\ & & \mid & \textbf{trylin}(E) \mid \textbf{commit}(p) \mid \langle \widetilde{C} \rangle \mid \widetilde{C}; \widetilde{C} \\ & & \mid & \textbf{if}\ (B)\ \widetilde{C}\ \textbf{else}\ \widetilde{C} \mid \textbf{while}\ (B)\{\widetilde{C}\} \\ (\textit{RelState}) & \Sigma & ::= & (\sigma, \Delta) \\ (\textit{SpecSet}) & \Delta & ::= & \{(U_1, \theta_1), \ldots, (U_n, \theta_n)\} \\ (\textit{PendThrds}) & U & ::= & \{\mathsf{t}_1 \rightsquigarrow \Upsilon_1, \ldots, \mathsf{t}_n \rightsquigarrow \Upsilon_n\} \\ (\textit{AbsOp}) & \Upsilon & ::= & (\gamma, n) \mid (\textbf{end}, n) \\ (\textit{RelAss}) & p, q, I & ::= & \mathsf{true} \mid \mathsf{false} \mid E = E \mid \mathsf{emp} \mid E \mapsto E \\ & & \mid & x \Mapsto E \mid E \rightarrowtail (\gamma, E) \mid E \rightarrowtail (\textbf{end}, E) \\ & & \mid & p * q \mid p \oplus q \mid p \vee q \mid \ldots \\ (\textit{RelAct}) & R, G & ::= & p \ltimes q \mid [p] \mid R * R \mid R \oplus R \mid \ldots\end{array}$$

Figure 8. Instrumented Code and Relational State Model

$$\bullet \stackrel{\text{def}}{=} \{(\emptyset, \emptyset)\} \qquad \text{where } \bullet \in \textit{SpecSet}$$

$$f \perp g \stackrel{\text{def}}{=} \textit{dom}(f) \cap \textit{dom}(g) = \emptyset$$

$$\Delta_1 \sharp \Delta_2 \stackrel{\text{def}}{=} U_1 \perp U_2 \wedge \theta_1 \perp \theta_2\,,\ \text{where } (U_1, \theta_1) \in \Delta_1 \wedge (U_2, \theta_2) \in \Delta_2$$

$$\Delta_1 * \Delta_2 \stackrel{\text{def}}{=} \{(U_1 \uplus U_2, \theta_1 \uplus \theta_2) \mid (U_1, \theta_1) \in \Delta_1 \wedge (U_2, \theta_2) \in \Delta_2\}$$

$$\begin{array}{l}\Sigma_1 * \Sigma_2 \stackrel{\text{def}}{=} (\sigma_1 \uplus \sigma_2, \Delta_1 * \Delta_2) \\ \qquad \text{where } \Sigma_1 = (\sigma_1, \Delta_1), \Sigma_2 = (\sigma_2, \Delta_2), \sigma_1 \perp \sigma_2 \text{ and } \Delta_1 \sharp \Delta_2\end{array}$$

$$\Sigma_1 \oplus \Sigma_2 \stackrel{\text{def}}{=} \begin{cases} (\sigma, \Delta_1 \cup \Delta_2) & \text{if } \Sigma_1 = (\sigma, \Delta_1) \text{ and } \Sigma_2 = (\sigma, \Delta_2) \\ \textit{undefined} & \text{otherwise}\end{cases}$$

$$\{\!\{E\}\!\}_{\sigma} \stackrel{\text{def}}{=} \begin{cases} [\![E]\!]_{\sigma} & \text{if } \textit{dom}(\sigma) = \textit{fv}(E) \\ \textit{undefined} & \text{otherwise}\end{cases}$$

$$\begin{array}{lll}(\sigma, \Delta) \models \mathsf{true} & \text{always holds} & \\ (\sigma, \Delta) \models \mathsf{false} & \text{never holds} & \\ (\sigma, \Delta) \models E_1 = E_2 & \text{iff} & \{\!\{(E_1 = E_2)\}\!\}_{\sigma} = \textbf{true} \wedge \Delta = \bullet \\ (\sigma, \Delta) \models \mathsf{emp} & \text{iff} & \sigma = \emptyset \wedge \Delta = \bullet \\ (\sigma, \Delta) \models E_1 \mapsto E_2 & \text{iff} & \exists l, n, \sigma'.\ \{\!\{(E_1, E_2)\}\!\}_{\sigma'} = (l, n) \\ & & \quad \wedge \sigma = \sigma' \uplus \{l \rightsquigarrow n\} \wedge \Delta = \bullet \\ (\sigma, \Delta) \models x \Mapsto E & \text{iff} & \exists n, \theta.\ \{\!\{E\}\!\}_{\sigma} = n \wedge \theta = \{x \rightsquigarrow n\} \\ & & \quad \wedge \Delta = \{(\emptyset, \theta)\} \\ (\sigma, \Delta) \models E_1 \rightarrowtail (\gamma, E_2) & \text{iff} & \exists \sigma_1, \sigma_2, \mathsf{t}, n.\ \sigma = \sigma_1 \uplus \sigma_2 \\ & & \quad \wedge \{\!\{E_1\}\!\}_{\sigma_1} = \mathsf{t} \wedge \{\!\{E_2\}\!\}_{\sigma_2} = n \\ & & \quad \wedge \Delta = \{(\{\mathsf{t} \rightsquigarrow (\gamma, n)\}, \emptyset)\} \\ (\sigma, \Delta) \models E_1 \rightarrowtail (\textbf{end}, E_2) & \text{iff} & \exists \sigma_1, \sigma_2, \mathsf{t}, n.\ \sigma = \sigma_1 \uplus \sigma_2 \\ & & \quad \wedge \{\!\{E_1\}\!\}_{\sigma_1} = \mathsf{t} \wedge \{\!\{E_2\}\!\}_{\sigma_2} = n \\ & & \quad \wedge \Delta = \{(\{\mathsf{t} \rightsquigarrow (\textbf{end}, n)\}, \emptyset)\}\end{array}$$

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

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

$$\mathsf{SpecExact}(p) \ \text{ iff } \ \forall \Delta, \Delta'.\ ((\_, \Delta) \models p) \wedge ((\_, \Delta') \models p) \Longrightarrow (\Delta = \Delta')$$

$$\mathsf{Exact}(p) \ \text{ iff } \ \forall \Sigma, \Sigma'.\ (\Sigma \models p) \wedge (\Sigma' \models p) \Longrightarrow (\Sigma = \Sigma')$$

$$\begin{array}{l}\mathsf{Precise}(p) \ \text{ iff} \\ \quad \forall \Sigma_1, \Sigma_2, \Sigma_1', \Sigma_2'.\ (\Sigma_1 * \Sigma_2 = \Sigma_1' * \Sigma_2') \wedge (\Sigma_1 \models p) \wedge (\Sigma_2 \models p) \\ \quad \Longrightarrow (\Sigma_1 = \Sigma_2)\end{array}$$

$$\mathsf{Sta}(p, R) \ \text{ iff } \ \forall \Sigma, \Sigma'.\ (\Sigma \models p) \wedge ((\Sigma, \Sigma') \models R) \Longrightarrow \Sigma' \models p$$

Figure 9. Semantics of State Assertions

4. A Relational Rely-Guarantee Style Logic

To prove object linearizability, we first instrument the object implementation by introducing auxiliary states and auxiliary commands, which relate the concrete code with the abstract object and operations. Our program logic extends LRG [8] with a relational interpretation of assertions and new rules for auxiliary commands. Although our logic is based on LRG [8], this approach is mostly in-

$$\frac{[E_1,p]\gamma[E_2,q]}{\vdash_{\mathsf{t}} \{\mathsf{t} \rightarrowtail (\gamma, E_1) * p\}\mathbf{linself}\{\mathsf{t} \rightarrowtail (\mathbf{end}, E_2) * q\}}\ (\textsc{linself}) \qquad \frac{}{\vdash_{\mathsf{t}} \{\mathsf{t} \rightarrowtail (\mathbf{end}, E)\}\mathbf{linself}\{\mathsf{t} \rightarrowtail (\mathbf{end}, E)\}}\ (\textsc{linself-end})$$

$$\frac{[E_1,p]\gamma[E_2,q]}{\vdash_{\mathsf{t}} \{E \rightarrowtail (\gamma, E_1) * p\}\mathbf{lin}(E)\{E \rightarrowtail (\mathbf{end}, E_2) * q\}}\ (\textsc{lin}) \qquad \frac{}{\vdash_{\mathsf{t}} \{E \rightarrowtail (\mathbf{end}, E')\}\mathbf{lin}(E)\{E \rightarrowtail (\mathbf{end}, E')\}}\ (\textsc{lin-end})$$

$$\frac{[E_1,p]\gamma[E_2,q]}{\vdash_{\mathsf{t}} \{\mathsf{t} \rightarrowtail (\gamma, E_1) * p\}\mathbf{trylinself}\{(\mathsf{t} \rightarrowtail (\gamma, E_1) * p) \oplus (\mathsf{t} \rightarrowtail (\mathbf{end}, E_2) * q)\}}\ (\textsc{tryself})$$

$$\frac{}{\vdash_{\mathsf{t}} \{\mathsf{t} \rightarrowtail (\mathbf{end}, E)\}\mathbf{trylinself}\{\mathsf{t} \rightarrowtail (\mathbf{end}, E)\}}\ (\textsc{tryself-end})$$

$$\frac{[E_1,p]\gamma[E_2,q]}{\vdash_{\mathsf{t}} \{E \rightarrowtail (\gamma, E_1) * p\}\mathbf{trylin}(E)\{(E \rightarrowtail (\gamma, E_1) * p) \oplus (E \rightarrowtail (\mathbf{end}, E_2) * q)\}}\ (\textsc{try})$$

$$\frac{}{\vdash_{\mathsf{t}} \{E \rightarrowtail (\mathbf{end}, E')\}\mathbf{trylin}(E)\{E \rightarrowtail (\mathbf{end}, E')\}}\ (\textsc{try-end}) \qquad \frac{\mathsf{SpecExact}(p) \qquad p' \Rightarrow p}{\vdash_{\mathsf{t}} \{p' \oplus \mathsf{true}\}\mathbf{commit}(p)\{p'\}}\ (\textsc{commit})$$

$$\frac{\vdash_{\mathsf{t}} \{p_1\}\mathbf{commit}(p)\{q\} \qquad p_2 \nmid p}{\vdash_{\mathsf{t}} \{p_1 \oplus p_2\}\mathbf{commit}(p)\{q\}}\ (\textsc{commit-spec-conj}) \qquad \frac{\begin{array}{c}\mathsf{Exact}(\{p_1,p_2\}) \qquad p_1 \oplus p_2 \text{ is satisfiable}\\ \vdash_{\mathsf{t}} \{p\}\mathbf{commit}(p_1)\{q_1\} \qquad \vdash_{\mathsf{t}} \{p\}\mathbf{commit}(p_2)\{q_2\}\end{array}}{\vdash_{\mathsf{t}} \{p\}\mathbf{commit}(p_1 \oplus p_2)\{q_1 \oplus q_2\}}\ (\textsc{multi-commit})$$

$$\frac{}{\vdash_{\mathsf{t}} \{\mathsf{t} \rightarrowtail (\mathbf{end}, E)\}\mathbf{E}[\,\mathbf{return}\ E\,]\{\mathsf{t} \rightarrowtail (\mathbf{end}, E)\}}\ (\textsc{ret}) \qquad \frac{\vdash_{\mathsf{t}} \{p\}\widetilde{C}\{q\}}{\vdash_{\mathsf{t}} \{p * r\}\widetilde{C}\{q * r\}}\ (\textsc{frame}) \qquad \frac{\vdash_{\mathsf{t}} \{p\}\widetilde{C}\{q\} \qquad \vdash_{\mathsf{t}} \{p'\}\widetilde{C}\{q'\}}{\vdash_{\mathsf{t}} \{p \oplus p'\}\widetilde{C}\{q \oplus q'\}}\ (\textsc{Spec-Conj})$$

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

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

$$\frac{R, G, I \vdash_{\mathsf{t}} \{p\}\widetilde{C}_1\{q\} \qquad R, G, I \vdash_{\mathsf{t}} \{q\}\widetilde{C}_2\{r\}}{R, G, I \vdash_{\mathsf{t}} \{p\}\widetilde{C}_1; \widetilde{C}_2\{r\}}\ (\textsc{p-seq}) \qquad \frac{R, G, I \vdash_{\mathsf{t}} \{p * B\}\widetilde{C}\{p * (B = B)\} \qquad p \Rightarrow I}{R, G, I \vdash_{\mathsf{t}} \{p * (B = B)\}\mathbf{while}\ (B)\{\widetilde{C}\}\{p * \neg B\}}\ (\textsc{p-while})$$

$$\frac{R, G, I \vdash_{\mathsf{t}} \{p * B\}\widetilde{C}_1\{q\} \qquad R, G, I \vdash_{\mathsf{t}} \{p * \neg B\}\widetilde{C}_2\{q\} \qquad p \Rightarrow I}{R, G, I \vdash_{\mathsf{t}} \{p * (B = B)\}\mathbf{if}\ (B)\ \widetilde{C}_1\ \mathbf{else}\ \widetilde{C}_2\{q\}}\ (\textsc{p-if})$$

$$\frac{R, G, I \vdash_{\mathsf{t}} \{p\}\widetilde{C}\{q\} \qquad p' \Rightarrow p \qquad q \Rightarrow q' \qquad R' \Rightarrow R \qquad G \Rightarrow G' \qquad p' \vee q' \Rightarrow I' * \mathsf{true} \qquad I' \triangleright \{R', G'\}}{R', G', I' \vdash_{\mathsf{t}} \{p'\}\widetilde{C}\{q'\}}\ (\textsc{p-conseq})$$

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

Figure 11. Selected Inference Rules

dependent with the base logic. Similar extensions can also be made over other logics, such as RGSep [31].

Our logic is proposed to verify object methods only. Verified object methods are guaranteed to be a contextual refinement of their abstract atomic operations, which ensures linearizability of the object. We discuss verification of whole programs consisting of both client code and object code at the end of Sec. 4.3.

4.1 Instrumented Code and States

In Fig. 8 we show the syntax of the instrumented code and its state model. As explained in Sec. 2, program states Σ for the object method executions now consist of two parts, the physical object states σ and the auxiliary data Δ. Δ is a *nonempty* set of (U, θ) pairs, each pair representing a speculation of the situation at the abstract level. Here θ is the current abstract object, and U is a pending thread pool recording the remaining operation to be fulfilled by each thread. It maps a thread id to its remaining abstract operation, which is either (γ, n) (the operation γ needs to be executed with argument n) or $(\mathbf{end}, n)$ (the operation has been finished with the return value n). We assume Δ is always *domain-exact*, defined as follows:

$$\mathsf{DomExact}(\Delta) \stackrel{\text{def}}{=} \forall U, \theta, U', \theta'.\ (U, \theta) \in \Delta \wedge (U', \theta') \in \Delta \\ \Longrightarrow \mathit{dom}(U) = \mathit{dom}(U') \wedge \mathit{dom}(\theta) = \mathit{dom}(\theta')\ .$$

It says, all the speculations in Δ should describe the same set of threads and the same domain of abstract objects. Any Δ containing a single speculation is domain-exact. Also domain-exactness can be preserved under the step of any command in our instrumented language, thus it is reasonable to assume it always holds.

$$(\Sigma, \Sigma') \models p \ltimes q \;\text{ iff }\; \Sigma \models p \wedge \Sigma' \models q$$
$$(\Sigma, \Sigma') \models [p] \;\text{ iff }\; \Sigma \models p \wedge \Sigma = \Sigma'$$
$$\begin{array}{l}(\Sigma, \Sigma') \models R_1 * R_2 \;\text{ iff}\\ \qquad\qquad \exists \Sigma_1, \Sigma_2, \Sigma_1', \Sigma_2'.\, (\Sigma = \Sigma_1 * \Sigma_2) \wedge (\Sigma' = \Sigma_1' * \Sigma_2') \\ \qquad\qquad\quad \wedge (\Sigma_1, \Sigma_1') \models R_1 \wedge (\Sigma_2, \Sigma_2') \models R_2\end{array}$$
$$\begin{array}{l}(\Sigma, \Sigma') \models R_1 \oplus R_2 \;\text{ iff}\\ \qquad\qquad \exists \Sigma_1, \Sigma_2, \Sigma_1', \Sigma_2'.\, (\Sigma = \Sigma_1 \oplus \Sigma_2) \wedge (\Sigma' = \Sigma_1' \oplus \Sigma_2') \\ \qquad\qquad\quad \wedge (\Sigma_1, \Sigma_1') \models R_1 \wedge (\Sigma_2, \Sigma_2') \models R_2\end{array}$$
$$\mathsf{Id} \overset{\text{def}}{=} [\mathsf{true}] \qquad \mathsf{Emp} \overset{\text{def}}{=} \mathsf{emp} \ltimes \mathsf{emp} \qquad \mathsf{True} \overset{\text{def}}{=} \mathsf{true} \ltimes \mathsf{true}$$
$$\frac{}{(\emptyset, n) \xrightarrow{\gamma} (\emptyset, n')} \qquad \frac{\gamma(n)(\theta) = (n', \theta') \quad (\Delta, n) \xrightarrow{\gamma} (\Delta', n')}{(\{(U, \theta)\} \uplus \Delta, n) \xrightarrow{\gamma} (\{(U, \theta')\} \uplus \Delta', n')}$$
$$\begin{array}{l}[E, p]\gamma[E', q] \;\text{ iff}\\ \qquad \forall \sigma, \Delta, n.\, (\sigma, \Delta) \models (E = n) * p \\ \qquad \Longrightarrow \;\exists \Delta', n'.\, (\Delta, n) \xrightarrow{\gamma} (\Delta', n') \wedge ((\sigma, \Delta') \models (E' = n') * q)\end{array}$$
$$I \triangleright R \;\text{ iff }\; ([I] \Rightarrow R) \wedge (R \Rightarrow I \ltimes I) \wedge \mathsf{Precise}(I)$$

Figure 10. Semantics of Actions

Below we informally explain the effects over Δ of the newly introduced commands. We leave their formal semantics to Sec. 4.4. The auxiliary command **linself** executes the unfinished abstract operation of the current thread in every U in Δ, and changes the abstract object θ correspondingly. **lin**(E) executes the abstract operation of the thread with id E. **linself** or **lin**(E) is executed when we know for sure that a step is the linearization point. The **trylinself** command introduces uncertainty. Since we do not know if the abstract operation of the current thread is fulfilled or not at the current point, we consider both possibilities. For each (U, θ) pair in Δ that contains unfinished abstract operation of the current thread, we add in Δ a new speculation (U', θ') where the abstract operation is done and θ' is the resulting abstract object. Since the original (U, θ) is also kept, we have both speculations in Δ. Similarly, the **trylin**(E) command introduces speculations about the thread E. When we have enough knowledge p about the situation of the abstract objects and operations, the **commit**(p) step keeps only the subset of speculations consistent with p and drops the rest. Here p is a logical assertion about the state Σ, which is explained below.

4.2 Assertions

Syntax of assertions is shown in Fig. 8. Following rely-guarantee style reasoning, assertions are either single state assertions p and q or binary rely/guarantee conditions R and G. Note here states refer to the relational states Σ.

We use standard separation logic assertions such as true, $E_1 = E_2$, emp and $E_1 \mapsto E_2$ to specify the memory σ. As shown in Fig. 9, their semantics is almost standard, but for $E_1 = E_2$ to hold over σ we require the domain of σ contains only the free variables in E_1 and E_2. Here we use $\llbrace E \rrbrace_\sigma$ to evaluate E with the extra requirement that σ contains the exact resource to do the evaluation.

New assertions are introduced to specify Δ. $x \Mapsto E$ specifies the abstract object θ in Δ, with no speculations of U (abstract operations), while $E_1 \rightarrowtail (\gamma, E_2)$ (and $E_1 \rightarrowtail (\mathbf{end}, E_2)$) specifies the singleton speculation of U. Semantics of separating conjunction $p * q$ is similar as in separation logic, except that it is now lifted to assertions over the relational states Σ. Note that the underlying "disjoint union" over Δ for separating conjunction should not be confused with the normal disjoint union operator over sets. The former (denoted as $\Delta_1 * \Delta_2$ in Fig. 9) describes the split of pending thread pools and/or abstract objects. For example, the left side Δ in the following equation specifies two speculations of threads $\mathtt{t}_1$ and $\mathtt{t}_2$ (we assume the abstract object part is empty and omitted here), and it can be split into two sets Δ_1 and Δ_2 on the right side, each of which describes the speculations of a single thread.

$$\left\{ \begin{array}{l|c|} \mathtt{t}_1 & \Upsilon_1 \\ \hline \mathtt{t}_2 & \Upsilon_2 \end{array}\,,\quad \begin{array}{l|c|} \mathtt{t}_1 & \Upsilon_1 \\ \hline \mathtt{t}_2 & \Upsilon_2' \end{array} \right\} = \begin{array}{c} \{\mathtt{t}_1\ \boxed{\Upsilon_1}\} \\ * \\ \{\mathtt{t}_2\ \boxed{\Upsilon_2},\quad \mathtt{t}_2\ \boxed{\Upsilon_2'}\} \end{array}$$

The most interesting new assertion is $p \oplus q$, where p and q specify two different speculations. It is this assertion that reflects uncertainty about the abstract level. However, the readers should not confuse $\oplus$ with disjunction. It is more like conjunction since it says Δ contains both speculations satisfying p *and* those satisfying q. As an example, the above equation could be formulated at the assertion level using $*$ and $\oplus$:

$$\begin{array}{rl} & (\mathtt{t}_1 \rightarrowtail \Upsilon_1 * \mathtt{t}_2 \rightarrowtail \Upsilon_2) \oplus (\mathtt{t}_1 \rightarrowtail \Upsilon_1 * \mathtt{t}_2 \rightarrowtail \Upsilon_2') \\ \Leftrightarrow & \mathtt{t}_1 \rightarrowtail \Upsilon_1 * (\mathtt{t}_2 \rightarrowtail \Upsilon_2 \oplus \mathtt{t}_2 \rightarrowtail \Upsilon_2') \end{array}$$

Rely and guarantee assertions specify transitions over Σ. Here we follow the syntax of LRG [8], with a new assertion $R_1 \oplus R_2$ specifying speculative behaviors of the environment. The semantics is given in Fig. 10. We will show the use of the assertions in the examples of Sec. 6.

4.3 Inference Rules

The rules of our logic are shown in Fig. 11. Rules on the top half are for sequential Hoare-style reasoning. They are proposed to verify code $\widetilde{C}$ in the atomic block $\langle \widetilde{C} \rangle$. The judgment is parameterized with the id $\mathtt{t}$ of the current thread.

For the **linself** command, if the abstract operation γ of the current thread has not been done, this command will finish it. Here $[E_1, p]\gamma[E_2, q]$ in the LINSELF rule describes the behavior of γ, which transforms abstract objects satisfying p to new ones satisfying q. E_1 and E_2 are the argument and return value respectively. The definition is given in Fig. 10. The LINSELF-END rule says **linself** has no effects if we know the abstract operation has been finished. The LIN rule and LIN-END rule are similar.

The TRY rule says that if the thread E has not finished the abstract operation γ, it can do speculation using **trylin**(E). The resulting state contains both cases, one says γ does not progress at this point and the other says it does. If the current thread has already finished the abstract operation, **trylin**(E) would have no effects, as shown in the TRY-END rule. The TRYSELF rule and TRYSELF-END rule are similar.

The above rules require us to know for sure either the abstract operation has been finished or not. If we want to support uncertainty in the pre-condition, we could first consider different cases and then apply the SPEC-CONJ rule, which is like the conjunction rule in traditional Hoare logic.

The COMMIT rule allows us to commit to a specific speculation and drop the rest. **commit**(p) keeps only the speculations satisfying p. We require p to describe an exact set of speculations, as defined by $\mathsf{SpecExact}(p)$ in Fig. 9. For example, the following p_1 is speculation-exact, while p_2 is not:

$$\begin{array}{l} p_1 \overset{\text{def}}{=} \mathtt{t} \rightarrowtail (\gamma, n) \oplus \mathtt{t} \rightarrowtail (\mathbf{end}, n') \\ p_2 \overset{\text{def}}{=} \mathtt{t} \rightarrowtail (\gamma, n) \vee \mathtt{t} \rightarrowtail (\mathbf{end}, n') \end{array}$$

In all of our examples in Sec. 6, the assertion p in **commit**(p) describes a singleton speculation, so $\mathsf{SpecExact}(p)$ trivially holds.

We also have the rules COMMIT-SPEC-CONJ and MULTI-COMMIT to handle more complex cases. The COMMIT-SPEC-CONJ allows to extend the precondition with some useless speculations satisfying p_2, where $p_2 \nmid p$ (defined in Figure 12) says the speculations satisfying p_2 should all be dropped for **commit**(p). The MULTI-COMMIT rule allows us to commit both speculations satisfying p_1 and those satisfying p_2, where p_1 and p_2 must be exact on *both* the concrete state and the speculation set (we define $\mathsf{Exact}(p)$

in Figure 9). The simple COMMIT rule is sufficient for all the examples we have verified, and we introduce the COMMIT-SPEC-CONJ and MULTI-COMMIT rules for interests only. We will discuss their possible use in Appendix B.

Before the current thread returns, it must know its abstract operation has been done, as required in the RET rule. We also have a standard FRAME rule as in separation logic for local reasoning.

Rules in the bottom half show how to do rely-guarantee style concurrency reasoning, which are very similar to those in LRG [8]. As in LRG, we use a precise invariant I to specify the boundary of the well-formed shared resource. The ATOM rule says we could reason sequentially about code in the atomic block. Then we can lift it to the concurrent setting as long as its effects over the shared resource satisfy the guarantee G, which is fenced by the invariant I. In this step we assume the environment does not update shared resource, thus using Id as the rely condition (see Fig. 10). To allow general environment behaviors, we should apply the ATOM-R rule later, which requires that R be fenced by I and the pre- and post-conditions be stable with respect to R. Here $\mathtt{Sta}(\{p, q\}, R)$ requires that p and q be stable with respect to R, a standard requirement in rely-guarantee reasoning.

To simplify the reasoning, we assume the return command **return** E occurs only at the end of the implementation code. It is not difficult to perform a pre-parser and transform any code to this form. In the example proofs in Sec. 6 and Appendix E, we sometimes still keep the original code although the reasoning is supposed to be done for transformed code.

Linking with client program verification. Our relational logic is introduced for object verification, but it can also be used to verify client code, since it is just an extension over the general-purpose concurrent logic LRG (which includes the rule for parallel composition). Moreover, as we will see in Sec. 5, our logic ensures contextual refinement. Therefore, to verify a program W, we could replace the object implementation with the abstract operations and verify the corresponding abstract program $\mathbb{W}$ instead. Since $\mathbb{W}$ abstracts away concrete object representation and method implementation details, this approach provides us with "separation and information hiding" [25] over the object, but still keeps enough information (*i.e.*, the abstract operations) about the method calls in concurrent client verification.

4.4 Semantics and Partial Correctness

We first show some key operational semantics rules for the instrumented code in Fig. 12.

A single step execution of the instrumented code by thread t is represented as $(\widetilde{C}, \Sigma) \hookrightarrow_{\mathsf{t}} (\widetilde{C}', \Sigma')$. When we reach the **return** E command (the second rule), we require that there be no uncertainty about thread t at the abstract level in Δ. That is, in every speculation in Δ, we always know t's operation has been finished with the same return value E. Meanings of the auxiliary commands have been explained before. Here we use the auxiliary definition $\Delta \to_{\mathsf{t}} \Delta'$ to formally define their transitions over Δ. The semantics of **commit**(p) requires p to be speculation-exact (see Fig. 9). Also it uses $(\sigma, \Delta)|_p = (\sigma', \Delta')$ to filter out the wrong speculations. To ensure locality, this filter allows Δ to contain some extra resource such as the threads and their abstract operations other than those described in p. For example, the following Δ describes two threads t_1 and t_2, but we could mention only t_1 in **commit**(p).

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

If p is $\mathsf{t}_1 \rightarrowtail (\gamma_1, n_1)$, then **commit**$(p)$ will keep only the left speculation and discard the other. p can also be $\mathsf{t}_1 \rightarrowtail (\gamma_1, n_1) \oplus \mathsf{t}_1 \rightarrowtail (\mathbf{end}, n_1')$, then **commit**$(p)$ will keep both speculations.

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

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

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

$$\frac{[\![E]\!]_\sigma = \mathsf{t}' \qquad \Delta \to_{\mathsf{t}'} \Delta'}{(\mathbf{E}[\,\mathbf{lin}(E)\,], (\sigma, \Delta)) \hookrightarrow_{\mathsf{t}} (\mathbf{E}[\,\mathbf{skip}\,], (\sigma, \Delta'))}$$

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

$$\frac{[\![E]\!]_\sigma = \mathsf{t}' \qquad \Delta \to_{\mathsf{t}'} \Delta'}{(\mathbf{E}[\,\mathbf{trylin}(E)\,], (\sigma, \Delta)) \hookrightarrow_{\mathsf{t}} (\mathbf{E}[\,\mathbf{skip}\,], (\sigma, \Delta \cup \Delta'))}$$

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

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

Auxiliary Definitions:

$$\frac{U(\mathsf{t}) = (\gamma, n) \qquad \gamma(n)(\theta) = (n', \theta')}{(U, \theta) \dashrightarrow_{\mathsf{t}} (U\{\mathsf{t} \rightsquigarrow (\mathbf{end}, n')\}, \theta')} \qquad \frac{U(\mathsf{t}) = (\mathbf{end}, n)}{(U, \theta) \dashrightarrow_{\mathsf{t}} (U, \theta)}$$

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

$(\sigma, \Delta)|_p = (\sigma', \Delta')$ iff
$\quad \exists \sigma'', \Delta'', \Delta_p.\ (\sigma = \sigma' \uplus \sigma'') \wedge (\Delta = \Delta' \uplus \Delta'') \wedge ((\sigma', \Delta_p) \models p)$
$\qquad\qquad \wedge (\Delta'|_{dom(\Delta_p)} = \Delta_p) \wedge (\Delta''|_{dom(\Delta_p)} \cap \Delta_p = \emptyset)$

$\Delta|_D \overset{\text{def}}{=} \{(U, \theta) \mid dom(\{(U, \theta)\}) = D \wedge \exists U', \theta'.\ (U \uplus U', \theta \uplus \theta') \in \Delta\}$

$dom(\Delta) \overset{\text{def}}{=} (dom(U), dom(\theta)) \qquad$ where $(U, \theta) \in \Delta$

$q \nmid p$ iff $\forall \Delta, \Delta_p.\ (\_, \Delta) \models q \wedge (\_, \Delta_p) \models p \implies (\Delta|_{dom(\Delta_p)} \cap \Delta_p = \emptyset)$

Figure 12. Operational Semantics in the Relational State Model

Given the thread-local semantics, we could next define the transition $(\widetilde{C}, \Sigma) \xhookrightarrow{R}_{\mathsf{t}} (\widetilde{C}, \Sigma)$, which describes the behavior of thread t with interference R from the environment.

Semantics preservation by the instrumentation. It is easy to see that the newly introduced auxiliary commands do not change the physical state σ, nor do they affect the program control flow. Thus the instrumentation does not change program behaviors, unless the auxiliary commands are inserted into the wrong places and they get stuck, but this can be prevented by our program logic.

Soundness w.r.t. partial correctness. Following LRG [8], we could give semantics of the logic judgment as $R, G, I \models_{\mathsf{t}} \{p\}\widetilde{C}\{q\}$, which encodes partial correctness of $\widetilde{C}$ *w.r.t.* the pre- and post-conditions. We could prove the logic ensures partial correctness by showing $R, G, I \vdash_{\mathsf{t}} \{p\}\widetilde{C}\{q\}$ implies $R, G, I \models_{\mathsf{t}} \{p\}\widetilde{C}\{q\}$. The details are shown in Appendix C. In the next section, we give a stronger soundness of the logic, *i.e.* soundness *w.r.t.* linearizability.

5. Soundness via Simulation

Our logic intuitively relates the concrete object code with its abstract level specification. In this section we formalize the intuition and prove that the logic indeed ensures object linearizability. The proof is constructed in the following steps. We propose a new rely-guarantee-based forward-backward simulation between the concrete code and the abstract operation. We prove the simulation is compositional and implies contextual refinement between the two sides, and our logic indeed establishes such a simulation. Thus the logic establishes contextual refinement. Finally we get linearizability following Theorem 6.

Below we first define a rely-guarantee-based forward-backward simulation. It extends RGSim [18] with the support of the helping mechanism and speculations.

Definition 7 (Simulation for Method). $(x, C) \preceq^{\mathsf{t}}_{R;G;p} \gamma$ iff

$$\forall n, \sigma, \Delta.\ (\sigma, \Delta) \models (\mathsf{t} \rightarrowtail (\gamma, n) * (x = n) * p) \\ \Longrightarrow\ (C; \textbf{noret}, \sigma) \preceq^{\mathsf{t}}_{R;G;p} \Delta\,.$$

Whenever $(C, \sigma) \preceq^{\mathsf{t}}_{R;G;p} \Delta$, we have the following:

1. if $C \neq \mathbf{E}[\,\textbf{return}\ \_\,]$, then
   (a) for any C' and σ', if $(C, \sigma) \longrightarrow_{\mathsf{t}} (C', \sigma')$, then there exists Δ' such that $\Delta \rightrightarrows \Delta'$, $((\sigma, \Delta), (\sigma', \Delta')) \models (G * \mathsf{True})$ and $(C', \sigma') \preceq^{\mathsf{t}}_{R;G;p} \Delta'$;
   (b) $(C, \sigma) \not\longrightarrow_{\mathsf{t}} \textbf{abort}$;
2. for any σ' and Δ', if $((\sigma, \Delta), (\sigma', \Delta')) \models (R * \mathsf{Id})$, then $(C, \sigma') \preceq^{\mathsf{t}}_{R;G;p} \Delta'$;
3. if $C = \mathbf{E}[\,\textbf{return}\ E\,]$, then there exists n' such that $[\![E]\!]_\sigma = n'$ and $(\sigma, \Delta) \models (\mathsf{t} \rightarrowtail (\textbf{end}, n') * (x = \_) * p)$.

As in RGSim, $(x, C) \preceq^{\mathsf{t}}_{R;G;p} \gamma$ says, the implementation C is simulated by the abstract operation γ under the interference with the environment, which is specified by R and G. The new simulation holds if the executions of the concrete code C are related to the *speculative* executions of some Δ. The Δ could specify abstract operations of other threads that might be helped, as well as the current thread t. Initially, the abstract operation of t is γ, with the same argument n as the concrete side (*i.e.*, $x = n$). The abstract operations of other threads can be known from the precondition p.

For each step of the concrete code C, we require it to be safe, and correspond to some steps of Δ, as shown in the first condition in Definition 7. We define the transition $\Delta \rightrightarrows \Delta'$ as follows.

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

where $(U, \theta) \dashrightarrow (U', \theta') \overset{\text{def}}{=} \exists \mathsf{t}.\ (U, \theta) \dashrightarrow_{\mathsf{t}} (U', \theta')$
and $(U, \theta) \dashrightarrow_{\mathsf{t}} (U', \theta')$ has been defined in Fig. 12.

It says, any (U', θ') pair in Δ' should be "reachable" from Δ. Specifically, we could execute the abstract operation of some thread t' (which could be the current thread t or some others), or drop some (U, θ) pair in Δ. The former is like a step of **trylin**(t') or **lin**(t'), depending on whether or not we keep the original abstract operation of t'. The latter can be viewed as a **commit** step, in which we discard the wrong speculations.

We also require the related steps at the two levels to satisfy the guarantee $G * \mathsf{True}$, G for the shared part and True (arbitrary transitions) for the local part. Symmetrically, the second condition in Definition 7 says, the simulation should be preserved under the environment interference $R * \mathsf{Id}$, R for the shared part and Id (identity transitions) for the local part.

Finally, when the method returns (the last condition in Definition 7), we require the current thread t has finished its abstract operation, and the return values match at the two levels.

Like RGSim, our new simulation is *compositional*, thus can ensure contextual refinement between the implementation and the abstract operation, as shown in the following lemma.

Lemma 8 (Simulation Implies Contextual Refinement).
For any Π, Γ and φ, if there exist R, G, p and I such that the following hold for all t,

1. for any f such that $\Pi(f) = (x, C)$, we have $\Pi(f) \preceq^{\mathsf{t}}_{R_{\mathsf{t}};G_{\mathsf{t}};p_{\mathsf{t}}} \Gamma(f)$, and $x \notin \mathit{dom}(I)$;
2. $R_{\mathsf{t}} = \bigvee_{\mathsf{t}' \neq \mathsf{t}} G_{\mathsf{t}'}$, $\ I \triangleright \{R_{\mathsf{t}}, G_{\mathsf{t}}\}$, $\ p_{\mathsf{t}} \Rightarrow I$, and $\mathsf{Sta}(p_{\mathsf{t}}, R_{\mathsf{t}})$;
3. $\lfloor \varphi \rfloor \Rightarrow \bigwedge_{\mathsf{t}} p_{\mathsf{t}}$;

then $\Pi \sqsubseteq_\varphi \Gamma$.

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

Figure 13. Erasure of Instrumented Code

Here $x \notin \mathit{dom}(I)$ means the formal argument x is always in the local state, and $\lfloor \varphi \rfloor$ lifts φ to a state assertion:

$$\lfloor \varphi \rfloor \overset{\text{def}}{=} \{(\sigma, \{(\emptyset, \theta)\}) \mid \varphi(\sigma) = \theta\}.$$

Lemma 8 allows us to prove contextual refinement $\Pi \sqsubseteq_\varphi \Gamma$ by showing the simulation $\Pi(f) \preceq^{\mathsf{t}}_{R_{\mathsf{t}};G_{\mathsf{t}};p_{\mathsf{t}}} \Gamma(f)$ for each method f, where R, G and p are defined over the shared states fenced by the invariant I, and the interference constraint $R_{\mathsf{t}} = \bigvee_{\mathsf{t}' \neq \mathsf{t}} G_{\mathsf{t}'}$ holds following Rely-Guarantee reasoning [17]. Its proof is similar to the compositionality proofs of RGSim [18], but now we need to be careful with the helping between threads and the speculations. We give the formal proofs in Appendix C.

Lemma 9 (Logic Ensures Simulation for Method).
For any t, x, C, γ, R, G and p, if there exist I and $\widetilde{C}$ such that

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

and $\mathsf{Er}(\widetilde{C}) = (C; \textbf{noret})$, then $(x, C) \preceq^{\mathsf{t}}_{R;G;p} \gamma$.

Here we use $\mathsf{Er}(\widetilde{C})$ to erase the instrumented commands in $\widetilde{C}$, which is defined in Figure 13. The lemma shows that, verifying $\widetilde{C}$ in our logic establishes simulation between the original code and the abstract operation. It is proved by first showing that our logic ensures the standard rely-guarantee-style partial correctness (see Sec. 4.4). Then we build the simulation by projecting the instrumented semantics (Fig. 12) to the concrete semantics of C (Fig. 5) and the speculative steps $\rightrightarrows$ of Δ.

Finally, from Lemmas 8 and 9, we get the soundness theorem of our logic, which says our logic can verify linearizability.

Theorem 10 (Logic Soundness). For any Π, Γ and φ, if there exist R, G, p and I such that the following hold for all t,

1. for any f, if $\Pi(f) = (x, C)$, there exists $\widetilde{C}$ such that
   $$R_{\mathsf{t}}, G_{\mathsf{t}}, I \vdash_{\mathsf{t}} \{\mathsf{t} \rightarrowtail (\Gamma(f), x) * p_{\mathsf{t}}\}\ \widetilde{C}\ \{\mathsf{t} \rightarrowtail (\textbf{end}, \_) * (x = \_) * p_{\mathsf{t}}\}\,,$$
   $\mathsf{Er}(\widetilde{C}) = (C; \textbf{noret})$, and $x \notin \mathit{dom}(I)$;
2. $R_{\mathsf{t}} = \bigvee_{\mathsf{t}' \neq \mathsf{t}} G_{\mathsf{t}'}$, $\ p_{\mathsf{t}} \Rightarrow I$, and $\mathsf{Sta}(p_{\mathsf{t}}, R_{\mathsf{t}})$;
3. $\lfloor \varphi \rfloor \Rightarrow \bigwedge_{\mathsf{t}} p_{\mathsf{t}}$;

then $\Pi \sqsubseteq_\varphi \Gamma$, and thus $\Pi \preceq_\varphi \Gamma$.

6. Examples

Our logic gives us an effective approach to verify linearizability. As shown in Table 1, we have verified 12 algorithms, including two stacks, three queues, four lists and three algorithms on atomic memory reads or writes. Table 1 summarizes their features, including the helping mechanism (**Helping**) and future-dependent LPs

Objects	Helping	Fut. LP	Java Pkg	HS Book
Treiber stack [28]				√
HSY stack [14]	√			√
MS two-lock queue [22]				√
MS lock-free queue [22]		√	√	√
DGLM queue [6]		√		
Lock-coupling list				√
Optimistic list [15]				√
Heller *et al.* lazy list [13]	√	√		√
Harris-Michael lock-free list	√	√	√	√
Pair snapshot [26]		√		
CCAS [30]	√	√		
RDCSS [12]	√	√		

Table 1. Verified Algorithms Using Our Logic

```
readPair(int i, j) {   local a, b, v, w;
  $\{I * (\mathsf{cid} \rightarrowtail (\gamma, (\mathtt{i}, \mathtt{j})))\}$
1 while(true) {
    $\{I * (\mathsf{cid} \rightarrowtail (\gamma, (\mathtt{i}, \mathtt{j})) \oplus \mathsf{true})\}$
2   < a := m[i].d; v := m[i].v; >
    $\{\exists v'. (I \wedge \mathsf{readCell}(\mathtt{i}, \mathtt{a}, \mathtt{v}; v')) * (\mathsf{cid} \rightarrowtail (\gamma, (\mathtt{i}, \mathtt{j})) \oplus \mathsf{true})\}$
3   < b := m[j].d; w := m[j].v;  trylinself; >
    $\{\exists v'. (I \wedge \mathsf{readCell}(\mathtt{i}, \mathtt{a}, \mathtt{v}; v') \wedge \mathsf{readCell}(\mathtt{j}, \mathtt{b}, \mathtt{w}; \_)) * \mathsf{afterTry}\}$
4   if (v = m[i].v) {
      $\{I * (\mathsf{cid} \rightarrowtail (\mathbf{end}, (\mathtt{a}, \mathtt{b})) \oplus \mathsf{true})\}$
5     commit(cid $\rightarrowtail$ (end, (a, b)));
      $\{I * (\mathsf{cid} \rightarrowtail (\mathbf{end}, (\mathtt{a}, \mathtt{b})))\}$
6     return (a, b);
      $\{I * (\mathsf{cid} \rightarrowtail (\mathbf{end}, (\mathtt{a}, \mathtt{b})))\}$
7   } } }
```

Auxiliary definitions:

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

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

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

Figure 14. Proof Outline of `readPair` in Pair Snapshot

(**Fut. LP**). Some of them are used in the `java.util.concurrent` package (**Java Pkg**). The last column (**HS Book**) shows whether it occurs in Herlihy and Shavit's classic textbook on concurrent algorithms [15]. We have almost covered all the fine-grained stacks, queues and lists in the book. We can see that our logic supports various objects ranging from simple ones with static LPs to sophisticated ones with non-fixed LPs. Although many of the examples can be verified using other approaches, we provide the first program logic which is proved sound and useful enough to verify all of these algorithms. The complete proofs of all the algorithms we have verified are given in Appendix E.

In general we verify linearizability in the following steps. First we instrument the code with the auxiliary commands such as **linself**, **trylin**(E) and **commit**(p) at proper program points. The instrumentation should not be difficult based on the intuition of the algorithm. Then, we specify the assertions (as in Theorem 10) and reason about the instrumented code by applying our inference rules, just like the usual partial correctness verification in LRG. In our experience, handling the auxiliary commands usually would not introduce much difficulty over the plain verification with LRG. Below we sketch the proofs of three representative examples: the pair snapshot, MS lock-free queue and the CCAS algorithm.

6.1 Pair Snapshot

As discussed in Sec. 2.3, the pair snapshot algorithm has a future-dependent LP. In Fig. 14, we show the proof of `readPair` for the current thread `cid`. We will use γ for its abstract operation, which atomically reads the cells `i` and `j` at the abstract level.

```
 1 enq(v) {                          16 deq() {
 2  local x, t, s, b;                17  local h, t, s, v, b;
 3  x := cons(v, null);              18  while (true) {
 4  while (true) {                   19   h := Head; t := Tail;
 5   t := Tail; s := t.next;         20   s := h.next;
 6   if (t = Tail) {                 21   if (h = Head)
 7    if (s = null) {                22    if (h = t) {
 8     b:=cas(&(t.next),s,x);        23     if (s = null)
 9     if (b) {                      24       return EMPTY;
10       cas(&Tail, t, x);           25     cas(&Tail, t, s);
11       return; }                   26    }else {
12    }else cas(&Tail, t, s);        27      v := s.val;
13   }                               28      b:=cas(&Head,h,s);
14  }                                29      if(b) return v; }
15 }                                 30  } }
```

Figure 15. MS Lock-Free Queue Code

First, we insert **trylinself** and **commit** as highlighted in Fig. 14. The **commit** command says, when the validation at line 4 succeeds, we must have $\mathtt{cid} \rightarrowtail (\mathbf{end}, (\mathtt{a}, \mathtt{b}))$ as a possible speculation. This actually requires a correct instrumentation of **trylinself**. In Fig. 14, we insert it at line 3. It cannot be moved to other program points since line 3 is the only place where we could get the abstract return value $(\mathtt{a}, \mathtt{b})$ when executing γ. Besides, we cannot replace it by a **linself**, because if line 4 fails later, we have to restart to do the original abstract operation.

After the instrumentation, we can define the precise invariant I, the rely R and the guarantee G. The invariant I simply maps every memory cell (d, v) at the concrete level to a cell with data d at the abstract level, as shown below:

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

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

Every thread guarantees that when writing a cell, it increases the version number. Here we use $[G]_I$ short for $(G \vee \mathsf{Id}) * \mathsf{Id} \wedge (I \ltimes I)$.

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

The rely R is the same as the guarantee G.

Then we specify the pre- and post-conditions, and reason about the instrumented code using our inference rules. The proof follows the intuition of the algorithm. Note that we relax $\mathtt{cid} \rightarrowtail (\gamma, (\mathtt{i}, \mathtt{j}))$ in the precondition of the method to $\mathtt{cid} \rightarrowtail (\gamma, (\mathtt{i}, \mathtt{j})) \oplus \mathsf{true}$ to ensure the loop invariant. The latter says, `cid` *may* just start (or restart) its operation and have not done yet.

The `readPair` method in the pair snapshot algorithm is "read-only" in the sense that the abstract operation does not update the abstract object. This perhaps means that it does not matter to linearize the method multiple times. In Sec. 6.3 we will verify an algorithm with future-dependent LPs, CCAS, which is not "read-only". We *can* still "linearize" a method with side effects multiple times.

6.2 MS Lock-Free Queue

The widely-used MS lock-free queue [22] also has future-dependent LPs. We show its code in Fig. 15.

The queue is implemented as a linked list with `Head` and `Tail` pointers. `Head` always points to the first node (a sentinel) in the list, and `Tail` points to either the last or second to last node. The `enq` method appends a new node at the tail of the list and advances `Tail`, and `deq` replaces the sentinel node by its next node and returns the value in the new sentinel. If the list contains only the sentinel node, meaning the queue is empty, then `deq` returns `EMPTY`.

The algorithm employs the helping mechanism for the `enq` method to swing the `Tail` pointer when it lags behind the end of the list. A thread should first try to help the half-finished `enq` by advancing `Tail` (lines 12 and 25 in Fig. 15) before doing its own operation. But this helping mechanism would not affect the LP of `enq` which is statically located at line 8 when the `cas` succeeds, since the new node already becomes visible in the queue after being appended to the list, and updating `Tail` will not affect the abstract queue. We simply instrument line 8 as follows to verify `enq`:

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

On the other hand, the original queue algorithm [22] checks `Head` or `Tail` (line 6 or 21 in Fig. 15) to make sure that its value has not been changed since its local copy was read (at line 5 or 19), and if it fails, the operation will restart. This check can improve efficiency of the algorithm, but it makes the LP of the `deq` method for the empty queue case depend on future executions. That LP should be at line 20, if the method returns `EMPTY` at the end of the same iteration. The intuition is, when we read `null` from `h.next` at line 20 (indicating the abstract queue must be empty there), we do not know how the iteration would terminate at that time. If the later check over `Head` at line 21 fails, the operation would restart and line 20 may not be the LP. We can use our try-commit instrumentation to handle this future-dependent LP. We insert **trylinself** at line 20, as follows:

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

Before the method returns `EMPTY`, we commit to the finished abstract operation, *i.e.*, we insert `commit(cid` ↣ `(end, EMPTY))` just before line 24. Also, when we know we have to do another iteration, we can commit to the original DEQ operation, *i.e.*, we insert `commit(cid` ↣ `DEQ)` at the end of the loop body.

For the case of nonempty queues, the LP of the `deq` method is statically at line 28 when the `cas` succeeds. Thus we can instrument **linself** there, as shown below.

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

After the instrumentation, we can define I, R and G and verify the code using our logic rules. The invariant I relates all the nodes in the concrete linked list to the abstract queue. R and G specify the related transitions at both levels, which simply include all the actions over the shared states in the algorithm. The proof is similar to the partial correctness proof using LRG, except that we have to specify the abstract objects and operations in assertions and reason about the instrumented code.

6.3 Conditional CAS

Conditional compare-and-swap (CCAS) [30] is a simplified version of the RDCSS algorithm [12]. It involves both the helping mechanism and future-dependent LPs. We show its code in Fig. 16.

The object contains an integer variable `a`, and a boolean bit `flag`. The method `SetFlag` (line 19) sets the bit directly. The method `CCAS` takes two arguments: an expected current value `o` of the variable `a` and a new value `n`. It atomically updates `a` with the new value if `flag` is true and `a` indeed has the value `o`; and does nothing otherwise. `CCAS` always returns the old value of `a`.

The implementation in Fig. 16 uses a variant of `cas`: instead of a boolean value indicating whether it succeeds, `cas(&a,o,n)` returns the old value stored in `a`. When starting a `CCAS`, a thread first allocates its descriptor (line 3), which contains the thread id and the arguments for CCAS. It then tries to put its descriptor in `a` (line 4). If successful (line 9), it calls the auxiliary `Complete` function, which restores `a` to the new value `n` (line 15) or to the original value `o` (line 17), depending on whether `flag` is true. If it finds `a` contains a descriptor (*i.e.*, `IsDesc` holds), it will try to help complete the operation in the descriptor (line 6) before doing its own. Since we disallow nested function calls to simplify the language, the auxiliary `Complete` function should be viewed as a macro.

The LPs of the algorithm are at lines 4, 7 and 13. If `a` contains a different value from `o` at lines 4 and 7, then CCAS fails and they are LPs of the current thread. We can instrument these lines as follows:

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

Figure 16. CCAS Code

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

If the descriptor `d` gets placed in `a`, then the LP should be in the `Complete` function. Since any thread can call `Complete` to help the operation, the LP should be at line 13 of the thread which will succeed at line 15 or 17. It is a future-dependent LP which may be in other threads' code. We instrument line 13 using **trylin**(`d.id`) to speculatively execute the abstract operation for the thread in `d`, which may not be the current thread. That is, line 13 becomes:

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

The condition `a=d` requires that the abstract operation in the descriptor has not been finished. Then at lines 15 and 17, we commit the correct guess. We show the instrumentation at line 15 below (where `s` is a local variable), and line 17 is instrumented similarly.

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

That is, it should be possible that the thread in `d` has finished the operation, and the current abstract `a` is the new value `n`.

Then we can define I, R and G, and verify the code by applying the inference rules. The invariant I says, the shared state includes `flag` and `a` at the abstract and the concrete levels; and when `a` is a descriptor `d`, the descriptor and the abstract operation of the thread `d.id` are also shared.

The rely R and the guarantee G should include the action over the shared states at each line. The action at line 4 (or 7) is interesting. If it succeeds, *both* the descriptor `d` and the abstract operation will be transferred from the local state to the shared part. This puts the abstract operation in the pending thread pool and enables other threads to help execute it.

The action at line 13 guarantees $\mathsf{TrylinSucc} \vee \mathsf{TrylinFail}$, which demonstrates the use of our logic for both helping and speculation.

$$\begin{aligned}\mathsf{TrylinSucc} &\overset{\text{def}}{=} (\exists t, o, n.\ (\mathtt{flag} \Mapsto \mathbf{true} * \mathsf{notDone}(t,o,n)) \\ &\qquad\qquad \ltimes (\mathtt{flag} \Mapsto \mathbf{true} * \mathsf{endSucc}(t,o,n))) \oplus \mathsf{Id} \\ \text{where } \mathsf{notDone}(t,o,n) &\overset{\text{def}}{=} t \rightarrowtail (\mathsf{CCAS}, o, n) * \mathtt{a} \Mapsto o \\ \mathsf{endSucc}(t,o,n) &\overset{\text{def}}{=} t \rightarrowtail (\mathbf{end}, o) * \mathtt{a} \Mapsto n\end{aligned}$$

$\mathsf{TrylinFail}$ is symmetric for the case when `flag` $\Mapsto$ **false**. Here we use $R \oplus \mathsf{Id}$ (defined in Fig. 9) to describe the action of **trylin**. It means, after the action we will keep the original state as well as the new state resulting from R as possible speculations. Also, in $\mathsf{TrylinSucc}$ and $\mathsf{TrylinFail}$, the current thread is allowed to help execute the abstract CCAS of some thread t.

The subtle part in the proof is to ensure that, no thread could cheat by imagining another thread's help. In any program point of CCAS, the environment may have done **trylin** and helped the operation. But whether the environment has helped it or not, the **commit** at line 15 or 17 cannot fail. This means, we should not confuse the two kinds of nondeterminism caused by speculation and by environment interference. The former allows us to discard wrong guesses, while for the latter, we should consider *all* possible environments (including none).

7. Related Work and Conclusion

In addition to the work mentioned in Sec. 1 and 2, there is a large body of work on linearizability verification. Here we only discuss the most closely related work that can handle non-fixed LPs.

Our logic is similar to Vafeiadis' extension of RGSep to prove linearizability [31]. He also uses abstract objects and abstract atomic operations as auxiliary variables and code. There are two key differences between the logics. First he uses prophecy variables to handle future-dependent LPs, but there has been no satisfactory semantics given for prophecy variables so far. We use the simple try-commit mechanism, whose semantics is straightforward. Second the soundness of his logic *w.r.t.* linearizability is not specified and proved. We address this problem by defining a new thread-local simulation as the meta-theory of our logic. As we explained in Sec. 2, defining such a simulation to support non-fixed LPs is one of the most challenging issues we have to solve. Although recently Vafeiadis develops an automatic verification tool [32] with formal soundness for linearizability, his new work can handle non-fixed LPs for *read-only* methods only, and cannot verify algorithms like HSY stack, CCAS and RDCSS in our paper.

Recently, Turon *et al.* [30] propose logical relations to verify fine-grained concurrency, which establish contextual refinement between the library and the specification. Underlying the model a similar simulation is defined. Our pending thread pool is proposed concurrently with their "spec thread pool", while the speculation idea in our simulation is borrowed from their work, which traces back to forward-backward simulation [20]. What is new here is a new program logic and the way we instrument code to do relational reasoning. The set of syntactic rules, including the try-commit mechanism to handle uncertainty, is much easier to use than the semantic logical relations to construct proofs. On the other hand, they support higher-order features, recursive types and polymorphism, while we focus on concurrency reasoning and use only a simple first-order language.

O'Hearn *et al.* [24] prove linearizability of an optimistic variant of the lazy set algorithm by identifying the "Hindsight" property of the algorithm. Their Hindsight Lemma provides a *non-constructive* evidence for linearizability. Although Hindsight can capture the insights of the set algorithm, it remains an open problem whether the Hindsight-like lemmas exist for other concurrent algorithms.

Colvin *et al.* [3] formally verify the lazy set algorithm using a combination of forward and backward simulations between automata. Their simulations are not thread-local, where they need to know the program counters of all threads. Besides, their simulations are specifically constructed for the lazy set only, while ours is more general in that it can be satisfied by various algorithms.

The simulations defined by Derrick *et al.* [4] are thread-local and general, but they require the operations with non-fixed LPs to be read-only, thus cannot handle the CCAS example. They also propose a backward simulation to verify linearizability [27]. Although the method is proved to be complete, it does not support thread-local verification and there is no program logic given.

Elmas *et al.* [7] prove linearizability by incrementally *rewriting* the fine-grained code to the atomic operation. They do not need to locate LPs. Their rules are based on left/right movers and program refinements, but not for Hoare-style reasoning as in our work.

There are also lots of model-checking based tools (*e.g.*, [19, 33]) for *checking* linearizability. For example, Vechev *et al.* [33] check linearizability with user-specified non-fixed LPs. Their method is not thread modular. To handle non-fixed LPs, they need users to instrument the code with enough information about the actions of other threads, which usually demands a priori knowledge about the number of threads running in parallel, as shown in their example. Besides, although their checker can detect un-linearizable code, it will not terminate for linearizable methods in general.

Conclusion. We propose a new program logic to verify linearizability of algorithms with non-fixed LPs. The logic extends LRG [8] with new rules for the auxiliary commands introduced specifically for linearizability proof. We also give a relational interpretation of asssertions and rely/guarantee conditions to relate concrete implementations with the corresponding abstract operations. Underlying the logic is a new thread-local simulation, which gives us contextual refinement. Linearizability is derived based on its equivalence to refinement. Both the logic and the simulation support reasoning about the helping mechanism and future-dependent LPs. As shown in Table 1, we have applied the logic to verify various classic algorithms.

Acknowledgments

We would like to thank Matthew Parkinson, Zhong Shao, Jan Hoffmann and anonymous referees for their suggestions and comments on earlier versions of this paper. This work is supported in part by grants from National Natural Science Foundation of China (NSFC) under Grant No. 61073040 and 61229201, the National Hi-Tech Research and Development Program of China (Grant No. 2012AA010901), and Program for New Century Excellent Talents in Universities (Grant No. NCET-2010-0984). Part of the work is done during Hongjin Liang's visit to Yale University in 2012-2013, which is supported by China Scholarship Council.

References

[1] M. Abadi and L. Lamport. The existence of refinement mappings. *Theor. Comput. Sci.*, 82(2):253–284, 1991.

[2] D. Amit, N. Rinetzky, T. Reps, M. Sagiv, and E. Yahav. Comparison under abstraction for verifying linearizability. In *CAV'07*.

[3] R. Colvin, L. Groves, V. Luchangco, and M. Moir. Formal verification of a lazy concurrent list-based set algorithm. In *CAV'06*.

[4] J. Derrick, G. Schellhorn, and H. Wehrheim. Verifying linearisability with potential linearisation points. In *FM'11*.

[5] J. Derrick, G. Schellhorn, and H. Wehrheim. Mechanically verified proof obligations for linearizability. *ACM TOPLAS*, 33(1):4, 2011.

[6] S. Doherty, L. Groves, V. Luchangco, and M. Moir. Formal verification of a practical lock-free queue algorithm. In *FORTE'04*.

[7] T. Elmas, S. Qadeer, A. Sezgin, O. Subasi, and S. Tasiran. Simplifying linearizability proofs with reduction and abstraction. In *TACAS'10*.

[8] X. Feng. Local rely-guarantee reasoning. In *POPL'09*.

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

[10] A. Gotsman and H. Yang. Linearizability with ownership transfer. In *CONCUR'12*.

[11] T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In *DISC'01*.

[12] T. L. Harris, K. Fraser, and I. A. Pratt. A practical multi-word compare-and-swap operation. In *DISC'02*.

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

[14] D. Hendler, N. Shavit, and L. Yerushalmi. A scalable lock-free stack algorithm. In *SPAA'04*.

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

[16] M. Herlihy and J. Wing. Linearizability: a correctness condition for concurrent objects. *ACM TOPLAS*, 12(3):463–492, 1990.

[17] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM TOPLAS*, 5(4):596–619, 1983.

[18] H. Liang, X. Feng, and M. Fu. A rely-guarantee-based simulation for verifying concurrent program transformations. In *POPL'12*.

[19] Y. Liu, W. Chen, Y. A. Liu, and J. Sun. Model checking linearizability via refinement. In *FM'09*.

[20] N. A. Lynch and F. W. Vaandrager. Forward and backward simulations: I. untimed systems. *Inf. Comput.*, 121(2):214–233, 1995.

[21] M. M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *SPAA'02*.

[22] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC'96*.

[23] P. W. O'Hearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, 2007.

[24] P. W. O'Hearn, N. Rinetzky, M. T. Vechev, E. Yahav, and G. Yorsh. Verifying linearizability with hindsight. In *PODC'10*, .

[25] P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *POPL'04*, .

[26] S. Qadeer, A. Sezgin, and S. Tasiran. Back and forth: Prophecy variables for static verification of concurrent programs. Tech Report.

[27] G. Schellhorn, H. Wehrheim, and J. Derrick. How to prove algorithms linearisable. In *CAV'12*.

[28] R. K. Treiber. System programming: coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center, 1986.

[29] A. Turon and M. Wand. A separation logic for refining concurrent objects. In *POPL'11*.

[30] A. Turon, J. Thamsborg, A. Ahmed, L. Birkedal, and D. Dreyer. Logical relations for fine-grained concurrency. In *POPL'13*.

[31] V. Vafeiadis. Modular fine-grained concurrency verification. Thesis.

[32] V. Vafeiadis. Automatically proving linearizability. In *CAV*, 2010.

[33] M. T. Vechev, E. Yahav, and G. Yorsh. Experience with model checking linearizability. In *SPIN'09*.

A. Proofs for Theorem 6 (Equivalence Between Linearizability and Contextual Refinement)

A.1 Contextual Refinement Implies Linearizability

To prove this direction, for each concurrent history H of the object o, we find a specific client W which can generate the observable behavior $\mathcal{B}$ when using Π, and $\mathcal{B}$ *approximates* H. By contextual refinement, we know $\mathcal{B}$ can also be produced by an abstract execution of W using Γ instead. Then we show the history H' generated by this abstract execution is just the legal sequential history we want to find: a completion of H is linearizable *w.r.t.* H'.

Suppose each local variable has a special integer value `UNDEF` if it has not been initialized. Also suppose clients could use an instruction $\mathbf{print}(f, n)$ to print out $(\mathtt{t}, \mathbf{out}, (f, n))$ atomically, where $\mathtt{t}$ is the current thread ID, f is a string and n is an integer. It is reasonable to allow a thread to print out its ID, since we can always distinguish different threads of a client.

Then we say $\mathcal{B}$ *approximates* H, denoted by $\mathcal{B} \approx H$, if each $(\mathtt{t}, \mathbf{out}, (f, n))$ in $\mathcal{B}$ corresponds to $(\mathtt{t}, f, n)$ in H, and each $(\mathtt{t}, \mathbf{out}, n)$ corresponds to $(\mathtt{t}, \mathbf{ok}, n)$.

$$\frac{}{\epsilon \approx \epsilon} \qquad \frac{\lambda \approx e \qquad \mathcal{B} \approx H}{\lambda :: \mathcal{B} \approx e :: H}$$

where $\lambda \approx e$ is defined by:

- $(\mathtt{t}, \mathbf{out}, (f, n)) \approx (\mathtt{t}, f, n)$;
- $(\mathtt{t}, \mathbf{out}, n) \approx (\mathtt{t}, \mathbf{ok}, n)$;
- $(\mathtt{t}, \mathbf{obj}, \mathbf{abort}) \approx (\mathtt{t}, \mathbf{obj}, \mathbf{abort})$.

On the concrete side, for any history H (generated by a client W), we could have an observable behavior $\mathcal{B}$ (generated by a client W') so that $\mathcal{B}$ approximates H. We construct W' using a transformation $\mathbf{T}$ on W as follows: each method call $x := f(E)$ of thread $\mathtt{t}$ is transformed to the following code $C_{\mathtt{t},x,f,E}$, and the program structure and other instructions are left unchanged.

$\mathbf{T}(W) \in$ *Prog* is defined inductively as follows:

$$\begin{array}{lcl}
\mathbf{T}(\mathbf{skip}) & \stackrel{\text{def}}{=} & \mathbf{skip} \\
\mathbf{T}(\mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n) & \stackrel{\text{def}}{=} & \mathbf{let}\ \Pi\ \mathbf{in}\ \mathbf{T}_1(C_1) \,\|\, \ldots \,\|\, \mathbf{T}_n(C_n)
\end{array}$$

$\mathbf{T}_{\mathtt{t}}(C) \in$ *Stmt* is inductively defined as follows:

$$\begin{array}{lcl}
\mathbf{T}_{\mathtt{t}}(\mathbf{skip}) & \stackrel{\text{def}}{=} & \mathbf{skip} \\
\mathbf{T}_{\mathtt{t}}(c) & \stackrel{\text{def}}{=} & c \\
\mathbf{T}_{\mathtt{t}}(x := f(E)) & \stackrel{\text{def}}{=} & C_{\mathtt{t},x,f,E} \\
\mathbf{T}_{\mathtt{t}}(\mathbf{return}\ E) & \stackrel{\text{def}}{=} & \mathbf{return}\ E \\
\mathbf{T}_{\mathtt{t}}(\mathbf{noret}) & \stackrel{\text{def}}{=} & \mathbf{noret} \\
\mathbf{T}_{\mathtt{t}}(\langle C \rangle) & \stackrel{\text{def}}{=} & \langle C \rangle \\
\mathbf{T}_{\mathtt{t}}(\mathbf{if}\ (B)\ C_1\ \mathbf{else}\ C_2) & \stackrel{\text{def}}{=} & \mathbf{if}\ (B)\ \mathbf{T}_{\mathtt{t}}(C_1)\ \mathbf{else}\ \mathbf{T}_{\mathtt{t}}(C_2) \\
\mathbf{T}_{\mathtt{t}}(\mathbf{while}\ (B)\{C\}) & \stackrel{\text{def}}{=} & \mathbf{while}\ (B)\{\mathbf{T}_{\mathtt{t}}(C)\}
\end{array}$$

$\mathbf{T}_{\mathtt{t}}(\mathbf{E}) \in$ *ExecContext* is inductively defined as follows:

$$\begin{array}{lcl}
\mathbf{T}_{\mathtt{t}}([\,]) & \stackrel{\text{def}}{=} & [\,] \\
\mathbf{T}_{\mathtt{t}}(\mathbf{E}; C) & \stackrel{\text{def}}{=} & \mathbf{T}_{\mathtt{t}}(\mathbf{E}); \mathbf{T}_{\mathtt{t}}(C)
\end{array}$$

$\mathbf{T}(\mathcal{S}) \in$ *PState* is defined as follows:

$$\begin{array}{lcl}
\mathbf{T}(\mathcal{S}) & \stackrel{\text{def}}{=} & \{(\sigma'_c, \sigma_o, \mathbf{T}(\mathcal{K})) \mid \mathcal{S} = (\sigma_c, \sigma_o, \mathcal{K}) \wedge \sigma'_c \in \mathbf{T}(\sigma_c)\} \\
\mathbf{T}(\sigma_c) & \stackrel{\text{def}}{=} & \{\sigma_c \uplus (\biguplus_{\mathtt{t} \in \mathit{ThrdID}} \{x_{\mathtt{t}} \leadsto n, y_{\mathtt{t}} \leadsto m\}) \mid n, m \in \mathit{Int}\}
\end{array}$$

$\mathbf{T}(\mathcal{K}) \in$ *ThrdPool* is defined as follows:

$$\mathbf{T}(\mathcal{K}) \stackrel{\text{def}}{=} \{\mathtt{t} \leadsto \mathbf{T}_{\mathtt{t}}(\kappa) \mid \mathcal{K}(\mathtt{t}) = \kappa\}$$

$$\mathbf{T}_{\mathtt{t}}(\kappa) \stackrel{\text{def}}{=} \begin{cases} \circ & \text{if } \kappa = \circ \\ (\sigma_l, x_{\mathtt{t}}, \mathbf{T}_{\mathtt{t}}(\mathbf{E})[\,C'_{\mathtt{t},x}\,]) & \text{if } \kappa = (\sigma_l, x, \mathbf{E}[\,\mathbf{skip}\,]) \end{cases}$$

where $C'_{\mathtt{t},x} \stackrel{\text{def}}{=} \mathbf{print}(x_{\mathtt{t}}); x := x_{\mathtt{t}}$

Figure 17. Correspondence for Code and States

$$C_{\mathtt{t},x,f,E} \stackrel{\text{def}}{=} \begin{array}{rl}
 & \text{local } x_{\mathtt{t}}, y_{\mathtt{t}}; \\
1 & y_{\mathtt{t}} := E; \\
2 & \mathbf{print}(f, y_{\mathtt{t}}); \\
3 & x_{\mathtt{t}} := f(y_{\mathtt{t}}); \\
4 & \mathbf{print}(x_{\mathtt{t}}); \\
5 & x := x_{\mathtt{t}};
\end{array}$$

Note that the argument to the method call is recorded in $x_{\mathtt{t}}$ and the return value is recorded in $y_{\mathtt{t}}$. Both of these variables are local to thread $\mathtt{t}$. When $x := f(E)$ in W goes one step, we let $C_{\mathtt{t},x,f,E}$ goes three steps (without interference from other threads) to the same method body, thus the first print-outed event and the invocation event are generated "atomically". Similarly, when the method body returns in W, we let W' goes three steps and generate the return and the last print-outed events "atomically". Thus the observable behavior could approximate the history. The formal inductive definition of $\mathbf{T}$ is given in Figure 17.

The following lemma says, there is an observable behavior that approximates the concrete history.

Lemma 11. For any W, σ_c, σ_o and H, if $H \in \mathcal{H}[\![W, (\sigma_c, \sigma_o)]\!]$, then there exist σ'_c and $\mathcal{B}$ such that $\sigma'_c \in \mathbf{T}(\sigma_c)$, $\mathcal{B} \approx H$ and $\mathcal{B} \in \mathcal{O}[\![\mathbf{T}(W), (\sigma'_c, \sigma_o)]\!]$.

Proof: From $H \in \mathcal{H}[\![W, (\sigma_c, \sigma_o)]\!]$, we know there exist $\mathcal{S}$, config (which could be a pair of code and state or **abort**) and H_1 such that $\mathcal{S} = \mathsf{init}(\sigma_c, \sigma_o)$, $(W, \mathcal{S}) \xrightarrow{H_1}{}^{k}$ config and $\mathsf{get\_hist}(H_1) = H$.

By Lemma 12 below, we know there exist $H_2, \mathcal{B}, \mathcal{S}'$ and config$'$ such that $\mathcal{S}' \in \mathbf{T}(\mathcal{S})$, $(\mathbf{T}(W), \mathcal{S}') \xrightarrow{H_2}{}^{*}$ config$'$, $\mathsf{get\_obsv}(H_2) = \mathcal{B}$ and $\mathcal{B} \approx H$.

Since $\mathcal{S}' \in \mathbf{T}(\mathcal{S})$ and $\mathcal{S} = \mathsf{init}(\sigma_c, \sigma_o)$, we know there exists σ'_c such that $\sigma'_c \in \mathbf{T}(\sigma_c)$ and $\mathcal{S}' = \mathsf{init}(\sigma'_c, \sigma_o)$. Thus we get the conclusion. ☐

Lemma 12. For all k, W_1, $\mathcal{S}_1$, $\mathcal{S}_2$, H_1 and config_1, if

1. $(W_1, \mathcal{S}_1) \overset{H_1}{\longmapsto}{}^k \mathsf{config}_1$;
2. $\mathcal{S}_2 \in \mathbf{T}(\mathcal{S}_1)$,

then

$$\exists \mathsf{config}_2, H_2.\ \mathsf{get\_obsv}(H_2) \approx \mathsf{get\_hist}(H_1) \\ \wedge\ (\mathbf{T}(W_1), \mathcal{S}_2) \overset{H_2}{\longmapsto}{}^* \mathsf{config}_2 .$$

Proof: By induction over k.
Base Case:

- $k = 0$. Trivial.
- $k = 1$ and $\mathsf{config}_1 = (\mathbf{skip}, \mathcal{S}_1)$. There must be no event generated. Trivial.
- $k = 1$ and $\mathsf{config}_1 = \mathbf{abort}$. Suppose the step is of the thread t.
  - The step produces an object abort event $(\mathsf{t}, \mathbf{obj}, \mathbf{abort})$. We know it must be a step inside the method body, thus it cannot be $x := f(E)$. The code of this step must be the same on the target and the source sides. By the state transformation in Figure 17, we know the target object state is the same as the source object state, and the local state σ_l of t is the same as the source. Thus the same step could be made on the target side and the same abort event could be generated.
  - The step produces a client abort.
    No history event is generated in H_1, so we simply let $\mathbf{T}(W_1)$ go zero step and $H_2 = \epsilon$. Thus $\mathsf{get\_obsv}(H_2) \approx \mathsf{get\_hist}(H_1)$.

Inductive Step: $k = n + 1$. We know

$$(W_1, \mathcal{S}_1) \overset{e}{\longmapsto} (W_1', \mathcal{S}_1')\ \wedge\ (W_1', \mathcal{S}_1') \overset{H_1'}{\longmapsto}{}^n \mathsf{config} .$$

Due to the induction hypothesis, we only need to prove the following:

$$\exists \mathcal{S}_2', H_2.\ (\mathbf{T}(W_1), \mathcal{S}_2) \overset{H_2}{\longmapsto}{}^* (\mathbf{T}(W'), \mathcal{S}_2') \\ \wedge\ \mathcal{S}_2' \in \mathbf{T}(\mathcal{S}_1')\ \wedge\ \mathsf{get\_obsv}(H_2) \approx \mathsf{get\_hist}(e) .$$

Proof sketch:

- If the first step of W_1 is from $x := f(E)$ to the method body of the thread t, suppose the initial source code of t is $\mathbf{E}[\,x := f(E)\,]$ and the initial source state is $(\sigma_c, \sigma_o, \circ)$, then by the operational semantics we know: the resulting code is $(C; \mathbf{noret})$ and the resulting state is $(\sigma_c, \sigma_o, \kappa)$, where $\Pi(f) = (y, C)$, $[\![E]\!]_{\sigma_c} = n$ and $\kappa = (\{y \rightsquigarrow n\}, x, \mathbf{E}[\,\mathbf{skip}\,])$. An invocation event (t, f, n) is generated.
  We know $\mathbf{T}_\mathsf{t}(\mathbf{E}[\,x := f(E)\,]) = \mathbf{T}_\mathsf{t}(\mathbf{E})[\,C_{\mathsf{t},x,f,E}\,]$. Since $\mathcal{S}_2 \in \mathbf{T}(\mathcal{S}_1)$, we know the initial state for the target code of t is $(\sigma_c', \sigma_o, \circ)$, where $\sigma_c' \in \mathbf{T}(\sigma_c)$. We let the target code go three steps.
  - The resulting target code is $(C; \mathbf{noret})$. Since C is the method body which does not contain method calls, we know it is $\mathbf{T}_\mathsf{t}(C; \mathbf{noret})$.
  - The target steps would not abort because σ_c' is an extension of σ_c, and thus $[\![E]\!]_{\sigma_c'} = n$. Then the resulting state $\mathcal{S}_2'$ is $(\sigma_c'', \sigma_o, \kappa')$, where
    $$\sigma_c'' = \sigma_c'\{y_\mathsf{t} \rightsquigarrow n\},\ \text{and} \\ \kappa' = (\{y \rightsquigarrow n\}, x_\mathsf{t}, \mathbf{T}_\mathsf{t}(\mathbf{E})[\,C_{\mathsf{t},x}'\,])$$
    Thus $\sigma_c'' = \mathbf{T}(\sigma_c)$ and $\kappa' = \mathbf{T}_\mathsf{t}(\kappa)$. Thus the resulting target state $\mathcal{S}_2' \in \mathbf{T}(\mathcal{S}_1')$.
  - The target three steps produce $H_2 = (\mathsf{t}, \mathbf{out}, (f, n)) :: (\mathsf{t}, f, n)$. Thus $\mathsf{get\_obsv}(H_2) \approx \mathsf{get\_hist}(e)$.
- If the first step of W_1 is a return of the thread t, and suppose the initial source code of t is $\mathbf{E}'[\,\mathbf{return}\, E\,]$, the initial source state is $(\sigma_c, \sigma_o, \kappa)$ and the current stack frame $\kappa = (\sigma_l, x, \mathbf{E}[\,\mathbf{skip}\,])$, then by the operational semantics, we know: the resulting source code is $\mathbf{E}[\,\mathbf{skip}\,]$ and the resulting state is $(\sigma_c', \sigma_o, \circ)$, where $[\![E]\!]_{\sigma_o \uplus \sigma_l} = n$, and $\sigma_c' = \sigma_c\{x \rightsquigarrow n\}$.
  We know $\mathbf{T}_\mathsf{t}(\mathbf{E}'[\,\mathbf{return}\, E\,]) = \mathbf{T}_\mathsf{t}(\mathbf{E}')[\,\mathbf{return}\, E\,]$. Since $\mathcal{S}_2 \in \mathbf{T}(\mathcal{S}_1)$, we know the initial state for the target code of t is $(\sigma_c'', \sigma_o, \kappa')$, where $\sigma_c'' \in \mathbf{T}(\sigma_c)$, and $\kappa' = (\sigma_l, x_\mathsf{t}, \mathbf{T}_\mathsf{t}(\mathbf{E})[\,C_{\mathsf{t},x}'\,])$. We let the target code go three steps.
  - The resulting target code is $\mathbf{T}_\mathsf{t}(\mathbf{E})[\,\mathbf{skip}\,]$, which is just $\mathbf{T}_\mathsf{t}(\mathbf{E}[\,\mathbf{skip}\,])$.
  - The target steps would not abort. The resulting state $\mathcal{S}_2'$ is $(\sigma_c''', \sigma_o, \circ)$, where $\sigma_c''' = \sigma_c''\{x \rightsquigarrow n, x_\mathsf{t} \rightsquigarrow \_, y_\mathsf{t} \rightsquigarrow \_\}$. Thus the resulting target state $\mathcal{S}_2' \in \mathbf{T}(\mathcal{S}_1')$.
  - The target three steps produce $H_2 = (\mathsf{t}, \mathbf{ok}, n) :: (\mathsf{t}, \mathbf{out}, n)$. Thus $\mathsf{get\_obsv}(H_2) \approx \mathsf{get\_hist}(e)$.
- If the first step of W_1 is a normal client step (the current call stack $\kappa = \circ$), suppose the source state is $\mathcal{S}_1 = (\sigma_c, \sigma_o, \mathcal{K})$, by the state transformation, we know the target state $\mathcal{S}_2 = (\sigma_c', \sigma_o, \mathbf{T}(\mathcal{K}))$, where σ_c' is an extension of σ_c, and the target stack frame of the thread t is $\circ$. By the operational semantic, we know there is a corresponding target step.
- If the first step of W_1 is a normal object step, by the state transformation in Figure 17, we know the target object state and local state in the call stack are the same as the source side. Thus by the operational semantic, we know there is a corresponding target step.

The formal proof needs stratified induction (according to the program structure). □

Then we prove the following lemma, which says, at the abstract side, the observable behavior is "linearizable" *w.r.t.* the abstract history.

Lemma 13. For any W, σ_c, θ and $\mathcal{B}$, if $\mathcal{B} \in \mathcal{O}[\![\mathbf{T}(W), (\sigma_c, \theta)]\!]$ and $\mathcal{B} \approx H_1$, then there exist H_c and H_2 such that $H_c \in \mathsf{completions}(H_1)$, $H_c \preceq_{\mathsf{lin}} H_2$, and $H_2 \in \mathcal{H}[\![\mathbf{T}(W), (\sigma_c, \theta)]\!]$.

Proof: From $\mathcal{B} \in \mathcal{O}[\![\mathbf{T}(W), (\sigma_c, \theta)]\!]$, we know there exist $\mathcal{S}$, config and H such that

$$\mathbb{S} = \mathsf{init}(\sigma_c, \theta)\,, \mathsf{get\_obsv}(H) = \mathcal{B}\,, (\mathbf{T}(W), \mathbb{S}) \overset{H}{\Longrightarrow}{}^* \mathsf{config} .$$

Let $H_2 = \mathsf{get\_hist}(H)$. Thus $H_2 \in \mathcal{H}[\![\mathbf{T}(W), (\sigma_c, \theta)]\!]$. We want to prove that

$$\text{Goal:}\quad \exists H_c.\ H_c \in \mathsf{completions}(H_1)\ \wedge\ H_c \preceq_{\mathsf{lin}} H_2 .$$

Constructing H_c and Proving Linearizability Condition 1: By the abstract operational semantics, we know that for any t, $\mathcal{B}|_\mathsf{t}$ and $H_2|_\mathsf{t}$ must satisfy one of the following:

1. $\mathcal{B}|_\mathsf{t} \approx H_2|_\mathsf{t}$; or
2. $\exists n.\ \mathcal{B}|_\mathsf{t} :: (\mathsf{t}, \mathbf{out}, n) \approx H_2|_\mathsf{t}$; or
3. $\exists f, n.\ \mathcal{B}|_\mathsf{t} \approx H_2|_\mathsf{t} :: (\mathsf{t}, f, n)$.

Since $\mathcal{B} \approx H_1$, we know, for any t, $H_1|_\mathsf{t}$ and $H_2|_\mathsf{t}$ must satisfy one of the following:

1. $H_1|_\mathsf{t} = H_2|_\mathsf{t}$; or
2. $\exists n.\ H_1|_\mathsf{t} :: (\mathsf{t}, \mathbf{ok}, n) = H_2|_\mathsf{t}$; or
3. $\exists f, n.\ H_1|_\mathsf{t} = H_2|_\mathsf{t} :: (\mathsf{t}, f, n)$.

We construct H_e as follows. For any t, if it is the above case 2, we append the corresponding return event at the end of H_1. Since $\mathsf{well\_formed}(H_1)$ and $\mathsf{well\_formed}(H_2)$, we could prove $\mathsf{well\_formed}(H_e)$. Thus $H_e \in \mathsf{extensions}(H_1)$. Also H_e satisfies: for any t, one of the following holds:

1. $H_e|_{\mathsf{t}} = H_2|_{\mathsf{t}}$; or
2. $\exists f, n.\ H_e|_{\mathsf{t}} = H_2|_{\mathsf{t}} :: (\mathsf{t}, f, n)$.

Let $H_c = \mathsf{truncate}(H_e)$. Thus $H_c \in \mathsf{completions}(H_1)$. Since $\forall \mathsf{t}.\ \mathsf{is\_res}(\mathsf{last}(H_2|_{\mathsf{t}})) \wedge \mathsf{seq}(H_2|_{\mathsf{t}})$, we could prove that for any t,

1. if $H_e|_{\mathsf{t}} = H_2|_{\mathsf{t}}$, then $H_c|_{\mathsf{t}} = H_e|_{\mathsf{t}}$;
2. if $H_e|_{\mathsf{t}} = H_2|_{\mathsf{t}} :: (\mathsf{t}, f, n)$, then $H_c|_{\mathsf{t}} = H_2|_{\mathsf{t}}$.

Thus $\forall \mathsf{t}.\ H_c|_{\mathsf{t}} = H_2|_{\mathsf{t}}$.

Proving Linearizability Condition 2: We informally show that the bijection π implicit in $\forall \mathsf{t}.\ H_c|_{\mathsf{t}} = H_2|_{\mathsf{t}}$ preserves the response-invocation order.

Let $H_c(i)$ be a response event in H_c and let $H_c(j)$ be an invocation event. Then $\pi(i)$ and $\pi(j)$ are the indices of $H_c(i)$ and $H_c(j)$ in H_2 respectively. Suppose $i < j$. By the construction of H_c from H_1, we know the same response and invocation events are in H_1, and the response happens before the invocation. Let i' and j' be the indices of these events in H_1. Then $i' < j'$. Since $\mathcal{B}' \approx H_1$, we know i' and j' are exactly the indices of the corresponding observable events in H_1, and $\mathcal{B}'(i')$ is a receive event and $\mathcal{B}'(j')$ is a send event. By the abstract operational semantics, we know in H, the history return event is before the history invocation event since the approximate receive event is before the approximate send event. Thus $\pi(i) < \pi(j)$. □

Finally, we get linearizability from contextual refinement, and Lemmas 11 and 13.

Theorem 14 (Contextual Refinement Implies Linearizability). If $\Pi \sqsubseteq_\varphi \Gamma$, then $\Pi \preceq_\varphi \Gamma$.

Proof: We need to prove that

$$\begin{array}{l}\forall n, C_1, \ldots, C_n, \sigma_c, \sigma_o, \theta, H.\\ H \in \mathcal{H}[\![(\textbf{let}\ \Pi\ \textbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_o)]\!] \wedge (\varphi(\sigma_o) = \theta)\\ \Longrightarrow \exists H_c, H'.\ H_c \in \mathsf{completions}(H) \wedge \Gamma \rhd (\theta, H') \wedge H_c \preceq_{\mathsf{lin}} H'\end{array}$$

By Lemma 11, we know:

$$\begin{array}{l}\exists \sigma_c', \mathcal{B}.\ \sigma_c' \in \mathbf{T}(\sigma_c) \wedge \mathcal{B} \approx H\\ \wedge\ \mathcal{B} \in \mathcal{O}[\![(\textbf{let}\ \Pi\ \textbf{in}\ \mathbf{T}_1(C_1) \,\|\, \ldots \,\|\, \mathbf{T}_n(C_n)), (\sigma_c', \sigma_o)]\!]\end{array}$$

By the definition of $\Pi \sqsubseteq_\varphi \Gamma$ (Definition 5), we know:

$$\mathcal{B} \in \mathcal{O}[\![(\textbf{with}\ \Gamma\ \textbf{do}\ \mathbf{T}_1(C_1) \,\|\, \ldots \,\|\, \mathbf{T}_n(C_n)), (\sigma_c', \theta)]\!]$$

By Lemma 13, we know

$$\begin{array}{l}\exists H_c, H'.\ H_c \in \mathsf{completions}(H) \wedge H_c \preceq_{\mathsf{lin}} H'\\ \quad \wedge\ H' \in \mathcal{H}[\![(\textbf{with}\ \Gamma\ \textbf{do}\ \mathbf{T}_1(C_1) \,\|\, \ldots \,\|\, \mathbf{T}_n(C_n)), (\sigma_c', \theta)]\!].\end{array}$$

By definition, we know

$$\Gamma \rhd (\theta, H') .$$

Thus we get the conclusion. □

A.2 Linearizability Implies Contextual Refinement

To prove this direction, we show that for any client W, if a concrete execution using Π generates an observable behavior $\mathcal{B}$ and a history H, where H is linearizable *w.r.t.* a legal sequential history H', then $\mathcal{B}$ can also be generated by an abstract execution of W using Γ, accompanied with the history H'. We construct the abstract execution of W from the concrete execution and the linearizability relation between H and H' as follows:

- For any client step in the concrete execution, we make the same step in the abstract execution. This could be managed because the effect of a client instruction depends only on the client state in our language model (Section 3), and the client states are always identical on the concrete and the abstract sides.
- If the client invokes a method of the object on the concrete side, we let the abstract client invoke the method as well.
- For any concrete step inside a method (a step after the client invocation but before the method **return**s), we let the abstract side go zero step.
- If the concrete step is a **return** of a method which produces a return event e_r in H, we locate e_r in H', and for every unprocessed events before e_r in H', execute the corresponding atomic method calls on the abstract side, following the order of H' (until e_r is also produced by the abstract client). And then return the current method at the abstract level.

To help locate the event e_r and ensure that the current abstract code and state are consistent with the unprocessed history events, we introduce two auxiliary *observable* events $\textbf{send}(\mathsf{t}, f, n)$ and $\textbf{recv}(\mathsf{t}, n')$ to produce at the invocation and return respectively. Then we use the whole event trace to distinguish whether a method call has been processed.

We first define two new semantics at the concrete and abstract levels with $\textbf{send}(\mathsf{t}, f, n)$ and $\textbf{recv}(\mathsf{t}, n')$ generated, and prove linearizability implies the contextual refinement which uses these two new semantics at the two levels (Lemma 19). Finally we prove this new-semantics contextual refinement implies the normal-semantics contextual refinement. This part is easy, since each execution of the normal semantics can correspond to an execution of the new semantics, and vice versa.

We show the new semantics in Figure 18. Here we overload the notations for the normal semantics.

Then we define a relation $\mathfrak{R}$ between the programs and states of the two levels, which is determined by the linearizability relation between histories. It takes three parameters: the past concrete event trace H_1, the future concrete event trace H_2 and the object specification Γ.

Definition 15. $(W, \mathcal{S})\, \mathfrak{R}_{H_1, H_2, \Gamma}\, (\mathbb{W}, \mathbb{S})$ iff there exist H_1', σ_c, σ_o, θ and $\mathcal{K}$ such that

1. (State relation) $\mathcal{S} = (\sigma_c, \sigma_o, \mathcal{K})$; $\mathbb{S} = (\sigma_c, \theta, \lfloor \mathcal{K} \rfloor)$;
2. (Code relation under H_1') $\mathbb{W} = \mathbf{A}_{\Gamma, \mathcal{K}, H_1'}(W)$ where the function $\mathbf{A}$ is defined in Figure 19,

and H_1' and θ satisfy the following: suppose we know the whole history (from the beginning to the end of the execution) H_e, then θ is the middle object of its linearization, and H_1' replaces the first half part in H_1 by the linearization. Formally, there exists H_e, H_{e1}, H_e', H_{e1}' such that

1. (H_{e1} is the prefix of H_e) $\mathsf{get\_hist}(H_1 :: H_2) = H_e$; $\mathsf{get\_hist}(H_1) = H_{e1}$;
2. (H_e' is a linearization of H_e) $\exists H_c.\ H_c \in \mathsf{completions}(H_e) \wedge H_c \preceq_{\mathsf{lin}} H_e'$;
3. (H_{e1}' is a linearization of H_{e1}) $\exists H_{c1}.\ H_{c1} \in \mathsf{completions}(H_{e1}) \wedge H_{c1} \preceq_{\mathsf{lin}} H_{e1}'$;
4. (θ is the middle object in H_e') $\exists H_{e2}'.\ H_e' = H_{e1}' :: H_{e2}' \ \wedge\ \Gamma \blacktriangleright (H_{e1}', \theta, H_{e2}')$;
5. (H_1' is H_1 with H_{e1} replaced by H_{e1}') $H_{e1}' = \mathsf{get\_hist}(H_1')$; $H_1' \backslash H_{e1}' = H_1 \backslash H_{e1}$; $\mathsf{well\_formed}(H_1')$.

Here we define $\Gamma \blacktriangleright (H, \theta', H')$ to mean θ' is the intermediate abstract object state between H and H' where both H and H' satisfy Γ. That is, each pair of invocation and immediate response events in H and H' is an allowed input-output pair following the specification Γ, with the abstract object being continuously changed from the initial one θ. We use $\Gamma \blacktriangleright (\theta, H)$ as a shorthand for $\Gamma \blacktriangleright (\epsilon, \theta, H)$.

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

$$\frac{\kappa = (\sigma_l, x, C) \qquad [\![E]\!]_{\sigma_l \uplus \sigma_o} = n' \qquad \sigma_c' = \sigma_c\{x \rightsquigarrow n'\}}{(\mathbf{E}[\,\textbf{return}\ E\,], (\sigma_c, \sigma_o, \kappa)) \xrightarrow{(\mathtt{t},\textbf{ok},n')::\textbf{recv}(\mathtt{t},n')}_{\mathtt{t},\Pi} (C, (\sigma_c', \sigma_o, , \circ))}$$

$$\frac{f \in \mathit{dom}(\Gamma) \qquad [\![E]\!]_{\sigma_c} = n \qquad x \in \mathit{dom}(\sigma_c) \qquad \mathit{ak} = (x, \mathbf{E}[\,\textbf{skip}\,])}{(\mathbf{E}[\,x := f(E)\,], (\sigma_c, \theta, \circ)) \circ\!\!\xrightarrow{\textbf{send}(\mathtt{t},f,n)}_{\mathtt{t},\Gamma} (\textbf{fexec}(f, n), (\sigma_c, \theta, \mathit{ak}))}$$

$$\frac{\Gamma(f)(n)(\theta) = (n', \theta')}{(\textbf{fexec}(f, n), (\sigma_c, \theta, \mathit{ak})) \circ\!\!\xrightarrow{(\mathtt{t},f,n)::(\mathtt{t},\textbf{ok},n')}_{\mathtt{t},\Gamma} (\textbf{fret}(n'), (\sigma_c, \theta', \mathit{ak}))}$$

$$\frac{\mathit{ak} = (x, C) \qquad \sigma_c' = \sigma_c\{x \rightsquigarrow n'\}}{(\textbf{fret}(n'), (\sigma_c, \theta, \mathit{ak})) \circ\!\!\xrightarrow{\textbf{recv}(\mathtt{t},n')}_{\mathtt{t},\Gamma} (C, (\sigma_c', \theta, \circ))}$$

Figure 18. Selected Rules of the New Operational Semantics with **send** and **recv** Events Generated

$\mathbf{A}_{\Gamma,\mathcal{K},H}(W)$ is defined inductively as follows:

$$\begin{array}{l} \mathbf{A}_{\Gamma,\mathcal{K},H}(\textbf{skip}) \stackrel{\text{def}}{=} \textbf{skip} \\ \mathbf{A}_{\Gamma,\mathcal{K},H}(\textbf{let}\ \Pi\ \textbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n) \stackrel{\text{def}}{=} \\ \qquad\qquad \textbf{with}\ \Gamma\ \textbf{do}\ \mathbf{A}_{\lfloor\mathcal{K}\rfloor(1),H|_1}(C_1) \,\|\, \ldots \,\|\, \mathbf{A}_{\lfloor\mathcal{K}\rfloor(n),H|_n}(C_n) \end{array}$$

$\mathbf{A}_{\kappa,H}(C)$ is defined as follows:

$$\mathbf{A}_{\kappa,H}(C) \stackrel{\text{def}}{=} \begin{cases} C & \text{if } \kappa = \circ \\ \textbf{fexec}(f, n) & \text{if } \kappa \neq \circ \text{ and } \mathsf{last}(H) = \textbf{send}(\mathtt{t}, f, n) \\ \textbf{fret}(n') & \text{if } \kappa \neq \circ \text{ and } \mathsf{last}(H) = (\mathtt{t}, \textbf{ok}, n') \end{cases}$$

$\lfloor\mathcal{K}\rfloor$ is defined as follows:

$$\begin{array}{l} \lfloor\mathcal{K}\rfloor \stackrel{\text{def}}{=} \{\mathtt{t} \rightsquigarrow \lfloor\kappa\rfloor \mid \mathcal{K}(\mathtt{t}) = \kappa\} \\ \lfloor\kappa\rfloor \stackrel{\text{def}}{=} \begin{cases} \circ & \text{if } \kappa = \circ \\ (x, C) & \text{if } \kappa = (\_, x, C) \end{cases} \end{array}$$

Figure 19. Code Abstraction

$$\Gamma \blacktriangleright (H, \theta', H') \stackrel{\text{def}}{=} \exists \theta.\, \Gamma \blacktriangleright (\theta, H, \theta', H')$$

$$\frac{\Gamma \blacktriangleright (\theta, H')}{\Gamma \blacktriangleright (\theta, \epsilon, \theta, H')}$$

$$\frac{(n', \theta') = \Gamma(f)(n)(\theta) \qquad \Gamma \blacktriangleright (\theta', H', \theta'', H'')}{\Gamma \blacktriangleright (\theta,\ (\mathtt{t}, f, n) :: (\mathtt{t}, \textbf{ok}, n') :: H', \theta'', H'')}$$

We use $\Gamma \blacktriangleright (H_{e1}', \theta, H_{e2}')$ to split the legal sequential history into two parts, H_{e1}' has already been generated, H_{e2}' needs to be generated in the future execution, and the intermediate state θ is used as the current abstract object state.

The following lemma says that, the histories generated in our abstract semantics are legal sequential histories.

Lemma 16. $\forall \Gamma, \theta, H.\, \Gamma \rhd (\theta, H) \implies \Gamma \blacktriangleright (\theta, H)$.

We also require H_1' to be *well-formed*, meaning that its observable events and history events are in a proper order, so that the event trace could be generated by an execution. For an event trace H, $\mathsf{well\_formed}(H)$ iff, for every thread, each send event is followed by a history invocation event, and then followed by a history response event and a receive event. The formal definition is similar to the well-formedness of a history, and omitted here. We could see that the event traces generated by the new semantics for closed programs (executed from out-of-method states) are all well-formed.

The following lemma says, linearizability ensures that the initial programs at the concrete and abstract sides are related by $\mathfrak{R}$.

Lemma 17. For any n, C_1, …, C_n, $\mathcal{S}$, $\mathbb{S}$, σ_c, σ_o, $\mathcal{K}$, θ, H and Γ, if

1. $\mathcal{S} = (\sigma_c, \sigma_o, \mathcal{K})$; $\mathbb{S} = (\sigma_c, \theta, \lfloor\mathcal{K}\rfloor)$; $\forall \mathtt{t}.\, \mathcal{K}(\mathtt{t}) = \circ$;
2. there exist H_e, H_c and H_e' such that $\mathsf{get\_hist}(H) = H_e$; $H_c \in \mathsf{completions}(H_e)$; $\Gamma \blacktriangleright (\theta, H_e')$; $H_c \preceq_{\mathsf{lin}} H_e'$,

then

$$(\textbf{let}\ \Pi\ \textbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n, \mathcal{S})\, \mathfrak{R}_{\epsilon,H,\Gamma}\, (\textbf{with}\ \Gamma\ \textbf{do}\ C_1 \,\|\, \ldots \,\|\, C_n, \mathbb{S})\,.$$

Proof: Immediate by Definition 15. □

With the above definitions, we could formalize the following lemma, which says that for any concrete execution where its history is linearizable, and starting from any abstract program and state which is related by $\mathfrak{R}$ to an intermediate program and state in the concrete execution, the abstract steps could generate the same observable behaviors as the concrete remaining steps. The lemma is proved by induction over the program steps.

Lemma 18. For any k, Π and Γ, W_0, $\mathcal{S}_0$, W, $\mathcal{S}$, H_0, H, config, $\mathbb{W}$ and $\mathbb{S}$, if

1. $(W_0, \mathcal{S}_0) \xmapsto{H_0}{}^{*} (W, \mathcal{S})$, where $\exists \mathcal{K}.\, \mathcal{S}_0 = (\_, \_, \mathcal{K}) \wedge \forall \mathtt{t}.\, \mathcal{K}(\mathtt{t}) = \circ$;
2. $(W, \mathcal{S}) \xmapsto{H}{}^{k} \mathsf{config}$;
3. $(W, \mathcal{S})\, \mathfrak{R}_{H_0,H,\Gamma}\, (\mathbb{W}, \mathbb{S})$,

then

$$\begin{array}{l} \exists \mathsf{config}', H'.\, (\mathbb{W}, \mathbb{S}) \circ\!\!\xrightarrow{H'}{}^{*} \mathsf{config}' \\ \qquad \wedge\ \mathsf{get\_obsv}(H') = \mathsf{get\_obsv}(H)\,. \end{array}$$

Proof: By induction over k.
Base Case:

- $k = 0$. Trivial.
- $k = 1$ and $\mathsf{config} = (\textbf{skip}, \mathcal{S})$. Trivial.
- $k = 1$ and $\mathsf{config} = \textbf{abort}$.
  - $H = (\mathtt{t}, \textbf{clt}, \textbf{abort})$. Thus the call stack is $\circ$. By the code abstraction function in Figure 19, we know the abstract code

is the same as the concrete code. Since the client states are identical on the two sides, the abstract client can also go one step to abort.

- $H = (\mathsf{t}, \textbf{obj}, \textbf{abort})$. It's impossible because the history is linearizable and cannot end with an abort event.

Inductive Step: $k = n + 1$.

$$(W, \mathcal{S}) \xmapsto{H_1} (W', \mathcal{S}') \ \wedge\ (W', \mathcal{S}') \xmapsto{H_2}{}^n \mathsf{config}$$

By the induction hypothesis, we only need to prove that

$$\begin{array}{l}\exists \mathbb{W}', \mathbb{S}', H_1'.\ (\mathbb{W}, \mathbb{S}) \overset{H_1'}{\Longrightarrow}{}^*_{,} (\mathbb{W}', \mathbb{S}') \\ \wedge\ (W', \mathcal{S}')\, \mathfrak{R}_{H_0::H_1, H_2, \Gamma}\, (\mathbb{W}', \mathbb{S}') \wedge \mathsf{get\_obsv}(H_1) = \mathsf{get\_obsv}(H_1')\end{array}$$

Below we let $H = H_0 :: H_1 :: H_2$, $H_e = \mathsf{get\_hist}(H)$, $H_{e0} = \mathsf{get\_hist}(H_0)$, $H_{e1} = \mathsf{get\_hist}(H_1)$, $H_{e2} = \mathsf{get\_hist}(H_2)$, the linearization of H_e is H_e', the linearization of H_{e0} is H_{e0}', $H_e' = H_{e0}' :: H_{e3}'$, $H_{e3}' = H_{e1}' :: H_{e2}'$, the abstract objects between H_{e0}', H_{e1}' and H_{e2}' are θ and θ', and the corresponding event trace of H_{e0}' is H_0'.

Proof sketch:

- If the first step of W is from $x := f(E)$ to the method body of the thread t, a send event $\textbf{send}(\mathsf{t}, f, n)$ and an invocation event (t, f, n) are generated. Suppose the concrete code of t is $\mathbf{E}[\, x := f(E)\,]$. The initial call stack of the thread is $\kappa = \circ$. By the code abstraction function in Figure 19, we know the abstract code of t is $\mathbf{E}[\, x := f(E)\,]$. We let it go one step. Then,
  - This abstract step does not abort. The same send event is generated: $H_1' = \textbf{send}(\mathsf{t}, f, n)$.
  - The resulting code on the abstract side is $\textbf{fexec}(f, n)$. We can prove that $(W', \mathcal{S}')\, \mathfrak{R}_{H_0::H_1, H_2, \Gamma}\, (\mathbb{W}', \mathbb{S}')$.
- If the first step of W is from $(\textbf{return}\ E)$ to the client code and the concrete call stack $\kappa \neq \circ$, a return event $e_r = (\mathsf{t}, \textbf{ok}, n')$ and a receive event $\textbf{recv}(\mathsf{t}, n')$ are generated. By the concrete operational semantics, we know the last event of $H_0|_{\mathsf{t}}$ must be an invocation event $e_i = (\mathsf{t}, f, n')$. Since $H_0' \backslash H_{e0}' = H_0 \backslash H_{e0}$; and $\mathsf{well\_formed}(H_0')$, we know there are two cases only:
  - If $\mathsf{last}(H_0'|_{\mathsf{t}})$ is a send event, we know $\mathsf{last}(H_0'|_{\mathsf{t}}) = \textbf{send}(\mathsf{t}, f, n')$, then the abstract code of t is $\textbf{fexec}(f, n')$. Thus e_r must be in H_{e3}'.
    Suppose H_{e3}' begins with $e_i^1, e_r^1, \ldots, e_i^k, e_r^k (= e_r)$. Their thread IDs are $\mathsf{t}_1, \ldots, \mathsf{t}_k (= \mathsf{t})$ respectively. Below we prove that the abstract codes of these threads are all **fexec**.
    - All of $e_i^1, \ldots, e_i^k$ are in H_{e0}.
      If e_i^j is not in H_{e0}, then we know in H_e, e_i^j is after e_r. Since H_e is linearizable *w.r.t.* H_e', we know in H_e', e_i^j is after e_r, which contradicts the fact that e_i^j is before e_r^k in H_{e3}'.
    - None of $e_r^1, \ldots, e_r^k$ are in H_{e0}.
      This is because they are not in H_{e0}', and H_{e0} is linearizable *w.r.t.* H_{e0}'.
    - $\forall j \in [1..k]$, its call stack is not $\circ$.
      By the first point, we know the send events $\lambda_s^1, \ldots, \lambda_s^k$ which correspond to $e_i^1, \ldots, e_i^k$ are in H_0, thus are in H_0'. By the second point, we know the receive events $\lambda_r^1, \ldots, \lambda_r^k$ which correspond to $e_r^1, \ldots, e_r^k$ are not in H_0. Thus the call stacks of $\mathsf{t}_1, \ldots, \mathsf{t}_k$ are not $\circ$.
    - $\forall j \in [1..k]$. $\mathsf{last}(H_0'|_{\mathsf{t}_j}) = \lambda_s^j$.
      We know λ_s^j is in $H_0'|_{\mathsf{t}_j}$ and $e_i^j = \mathsf{last}(H_0|_{\mathsf{t}_j})$. If $\mathsf{last}(H_0'|_{\mathsf{t}_j}) \neq \lambda_s^j$, since $\mathsf{get\_obsv}(H_0') = \mathsf{get\_obsv}(H_0)$, we know the last event of $H_0'|_{\mathsf{t}_j}$ must be a history event. Then we could get a contradiction.

    We let the abstract client code be executed several steps as follows: for the threads $\mathsf{t}_1, \ldots, \mathsf{t}_k$, each executes one step to **fret** in order, and then t_k executes one step more to the client code. Then,
    - We have $H_1' = (e_i^1 :: e_r^1 :: \ldots :: e_i^k :: e_r^k :: \textbf{recv}(\mathsf{t}, n'))$ where $e_r^k = e_r$. This is because these events are at the beginning of H_{e3}', and $\Gamma \blacktriangleright (\theta, H_{e3}')$.
    - We can prove that $(W', \mathcal{S}')\, \mathfrak{R}_{H_0::H_1, H_2, \Gamma}\, (\mathbb{W}', \mathbb{S}')$.
  - If $\mathsf{last}(H_0'|_{\mathsf{t}}) = e_r' = (\mathsf{t}, \textbf{ok}, n_1')$, then the abstract code of t is $\textbf{fret}(n_1')$. We let it go one step. Then,
    - We can prove $e_r' = e_r$. Since $H_{c0} \preceq_{\mathsf{lin}} H_{e0}'$, we know $\mathsf{last}(H_{c0}|_{\mathsf{t}}) = e_r'$, and e_r' is not in H_{e0}. By the history completion operation, we know e_i must be in H_{c0}, thus is also in H_{e0}'. Thus we know e_i and e_r' are the last events in $H_{e0}'|_{\mathsf{t}}$. On the other hand, we have e_r is in $H_e|_{\mathsf{t}}$ and just follows e_i. Since H_e is linearizable *w.r.t.* H_e', we know e_r must be in $H_e'|_{\mathsf{t}}$ and also follows e_i. Since $H_{e0}'|_{\mathsf{t}}$ is a part of $H_e'|_{\mathsf{t}}$, we could conclude $e_r = e_r'$, and thus $n_1' = n'$.
    - The abstract step generates a receive event $\textbf{recv}(\mathsf{t}, n')$.
    - We can prove that $(W', \mathcal{S}')\, \mathfrak{R}_{H_0::H_1, H_2, \Gamma}\, (\mathbb{W}', \mathbb{S}')$.
- If the first step of W_1 is a normal step of a method (*i.e.*, the call stack is not $\circ$ and it is not returning), no event is generated. We let the abstract code go zero step.
- If the first step of W_1 is a normal step of the client (*i.e.*, the call stack is $\circ$ and it is not calling a method), no history event is generated but a user event might be generated. Then we can let the abstract code go the same step, since the client states are the same on the two sides and the semantics of the statements depend on only the client states.

From the induction hypothesis and the above argument that the first steps can generate the same observable behavior, we could finish the proof. □

Lemma 19. If $\Pi \preceq_\varphi \Gamma$, then $\Pi \sqsubseteq_\varphi \Gamma$.

Proof: For any $n, C_1, \ldots, C_n, \sigma_c, \sigma_o, \theta$ and $\mathcal{B}$, if $\mathcal{B} \in \mathcal{O}[\![(\textbf{let}\ \Pi\ \textbf{in}\ C_1 \parallel \ldots \parallel C_n), (\sigma_c, \sigma_o)]\!]$ and $\varphi(\sigma_o) = \theta$, we know there exist config, H and H_e such that

$$\begin{array}{c}(\textbf{let}\ \Pi\ \textbf{in}\ C_1 \parallel \ldots \parallel C_n, \mathcal{S}) \xmapsto{H}{}^* \mathsf{config}\,, \\ \mathcal{S} = \mathsf{init}(\sigma_c, \sigma_o)\,,\ \mathsf{get\_hist}(H) = H_e\,,\ \mathsf{get\_obsv}(H) = \mathcal{B}\,.\end{array}$$

Thus $H_e \in \mathcal{H}[\![(\textbf{let}\ \Pi\ \textbf{in}\ C_1 \parallel \ldots \parallel C_n), (\sigma_c, \sigma_o)]\!]$. From $\Pi \preceq_\varphi \Gamma$, we know

$$\exists H_c, H_e'.\ H_c \in \mathsf{completions}(H_e) \wedge \Gamma \triangleright (\theta, H_e') \wedge H_c \preceq_{\mathsf{lin}} H_e'\,.$$

By Lemma 16, we know $\Gamma \blacktriangleright (\theta, H_e')$. From Lemma 17, we know

$$\begin{array}{l}\exists \mathbb{S}.\ \mathbb{S} = \mathsf{init}(\sigma_c, \theta) \\ \wedge\ (\textbf{let}\ \Pi\ \textbf{in}\ C_1 \parallel \ldots \parallel C_n, \mathcal{S})\, \mathfrak{R}_{\epsilon, H, \Gamma}\, (\textbf{with}\ \Gamma\ \textbf{do}\ C_1 \parallel \ldots \parallel C_n, \mathbb{S})\,.\end{array}$$

By Lemma 18, we know
$\mathcal{B} \in \mathcal{O}[\![(\textbf{with}\ \Gamma\ \textbf{do}\ C_1 \parallel \ldots \parallel C_n), (\sigma_c, \theta)]\!]$. □

B. Inference Rules for Assertions and Actions and More Discussions on Commit

B.1 Assertions

Properties on the separating conjunction still hold. We list the inference rules in Separation Logic for separating conjunction below.

$$p * q \Leftrightarrow q * p$$
$$(p * q) * r \Leftrightarrow p * (q * r)$$
$$p * \mathsf{emp} \Leftrightarrow p$$
$$(p \vee p') * q \Leftrightarrow (p * q) \vee (p' * q)$$
$$(p \wedge p') * q \Rightarrow (p * q) \wedge (p' * q)$$
$$\text{If } \mathsf{Precise}(q),\ \text{then } (p * q) \wedge (p' * q) \ \Rightarrow\ (p \wedge p') * q\,.$$
$$\frac{p \Rightarrow p' \qquad q \Rightarrow q'}{p * q \Rightarrow p' * q'}\ \text{(monotonicity)}$$

Properties for the speculative conjunction.

(1) Commutativity.
$$p \oplus q \ \Leftrightarrow\ q \oplus p$$

(2) Associativity.
$$(p \oplus q) \oplus r \ \Leftrightarrow\ p \oplus (q \oplus r)$$

(3) Monotonicity.
$$\frac{p \Rightarrow p' \qquad q \Rightarrow q'}{p \oplus q \Rightarrow p' \oplus q'}$$

(4) Distributivity over $\vee$.
$$(p \vee p') \oplus q \ \Leftrightarrow\ (p \oplus q) \vee (p' \oplus q)$$

(5) Semi-distributivity over $\wedge$.
$$(p \wedge p') \oplus q \ \Rightarrow\ (p \oplus q) \wedge (p' \oplus q)$$

(6) Semi-idempotence.
$$p \Rightarrow p \oplus p$$
$$\mathsf{true} \Leftrightarrow \mathsf{true} \oplus \mathsf{true}$$

(7) $\wedge$-like property.
$$p \Rightarrow (q \Rightarrow (p \oplus q))$$
$$(p \wedge q) \Rightarrow (p \oplus q)$$
$$\mathsf{false} \Leftrightarrow p \oplus \mathsf{false}$$

(8) Semi-distributivity of $\vee$ over $\oplus$.
$$(p \oplus p') \vee q \ \Rightarrow\ (p \vee q) \oplus (p' \vee q)$$

(9) Semi-distributivity of $*$ over $\oplus$.
$$(p \oplus p') * q \ \Rightarrow\ (p * q) \oplus (p' * q)$$

(10) For assertions which are $\mathsf{SpecExact}$ and Exact,
$$\text{If } \mathsf{SpecExact}(p),\ \text{ then } p \oplus p \Rightarrow p\,.$$
$$\text{If } \mathsf{Exact}(q),\ \text{ then } (p * q) \oplus (p' * q) \ \Rightarrow\ (p \oplus p') * q\,.$$

Properties that do not hold and counterexamples.

(1) $\wedge$-like or $\vee$-like properties.
$$p \oplus q \Rightarrow p$$
$$p \Rightarrow p \oplus q$$
$$(p \Rightarrow r) \Rightarrow ((q \Rightarrow r) \Rightarrow ((p \oplus q) \Rightarrow r))$$

(2) Reverse direction of sound properties.
$$p \oplus p \ \Rightarrow\ p$$

Counterexample: Let $p = \mathtt{t} \rightarrowtail (\gamma, n) \vee \mathtt{t} \rightarrowtail (\mathbf{end}, n')$. The left side can be satisfied when $\Delta = \{(\{\mathtt{t} \leadsto (\gamma, n)\}, \emptyset), (\{\mathtt{t} \leadsto (\mathbf{end}, n')\}, \emptyset)\}$, but the right side cannot.

$$(p \oplus q) \wedge (p' \oplus q) \ \Rightarrow\ (p \wedge p') \oplus q$$

Counterexample: Let $p = \mathtt{t} \rightarrowtail (\gamma, n)$, $p' = \mathtt{t} \rightarrowtail (\mathbf{end}, n')$ and $q = \mathtt{t} \rightarrowtail (\gamma, n) \vee \mathtt{t} \rightarrowtail (\mathbf{end}, n')$. Then the righthand side is false. The left side can be satisfied when $\Delta = \{(\{\mathtt{t} \leadsto (\gamma, n)\}, \emptyset), (\{\mathtt{t} \leadsto (\mathbf{end}, n')\}, \emptyset)\}$. Another similar counterexample is when $q = \mathtt{t} \rightarrowtail (\gamma, n) \oplus \mathtt{t} \rightarrowtail (\mathbf{end}, n')$.

$$(p \vee q) \oplus (p' \vee q) \ \Rightarrow\ (p \oplus p') \vee q$$

Counterexample: Let $p = p' = \mathtt{t} \rightarrowtail (\gamma, n)$ and $q = \mathtt{t} \rightarrowtail (\mathbf{end}, n')$. The left side can be satisfied when $\Delta = \{(\{\mathtt{t} \leadsto (\gamma, n)\}, \emptyset), (\{\mathtt{t} \leadsto (\mathbf{end}, n')\}, \emptyset)\}$, but the right side cannot. (We have many counterexamples here.)

$$(p * q) \oplus (p' * q) \ \Rightarrow\ (p \oplus p') * q$$

Counterexample: Let $p = \mathtt{t}_1 \rightarrowtail (\gamma_1, n_1)$, $p' = \mathtt{t}_1 \rightarrowtail (\mathbf{end}, n_1')$ and $q = \mathtt{t}_2 \rightarrowtail (\gamma_2, n_2) \vee \mathtt{t}_2 \rightarrowtail (\mathbf{end}, n_2')$. Then the left side can be satisfied when $\Delta = \{(\{\mathtt{t}_1 \leadsto (\gamma_1, n_1), \mathtt{t}_2 \leadsto (\gamma_2, n_2)\}, \emptyset), (\{\mathtt{t}_1 \leadsto (\mathbf{end}, n_1'), \mathtt{t}_2 \leadsto (\mathbf{end}, n_2')\}, \emptyset)\}$, but the right side cannot.

$$(p * q) \oplus (p' * q) \Rightarrow (p \oplus p') * (q \oplus q)$$

Counterexample: Let $p = \mathtt{t}_1 \rightarrowtail (\gamma_1, n_1)$, $p' = \mathtt{t}_2 \rightarrowtail (\gamma_2, n_2)$ and $q = \mathtt{t}_1 \rightarrowtail (\gamma_1, n_1) \vee \mathtt{t}_2 \rightarrowtail (\gamma_2, n_2)$. Then the left side can be satisfied when $\Delta = \{(\{\mathtt{t}_1 \leadsto (\gamma_1, n_1), \mathtt{t}_2 \leadsto (\gamma_2, n_2)\}, \emptyset)\}$, but the right side is false. Note that it is irrelevant to whether q is precise or not. We can let $q = \mathtt{t}_1 \rightarrowtail (\gamma_1, n_1) * \mathtt{t}_3 \rightarrowtail (\gamma_3, n_3) \vee \mathtt{t}_2 \rightarrowtail (\gamma_2, n_2) * \mathtt{t}_3 \rightarrowtail (\mathbf{end}, n_3')$, which is precise, but it is still a counterexample.

(3) Distributivity of $\oplus$ over $*$.
$$(p * p') \oplus q \ \Rightarrow\ (p \oplus q) * (p' \oplus q)$$

Counterexample: Let $p = \mathtt{t}_1 \rightarrowtail (\gamma_1, n_1)$, $p' = \mathtt{t}_2 \rightarrowtail (\gamma_2, n_2)$ and $q = \mathtt{t}_1 \rightarrowtail (\mathbf{end}, n_1') * \mathtt{t}_2 \rightarrowtail (\mathbf{end}, n_2')$. Then the left side can be satisfied when $\Delta = \{(\{\mathtt{t}_1 \leadsto (\gamma_1, n_1), \mathtt{t}_2 \leadsto (\gamma_2, n_2)\}, \emptyset), (\{\mathtt{t}_1 \leadsto (\mathbf{end}, n_1'), \mathtt{t}_2 \leadsto (\mathbf{end}, n_2')\}, \emptyset)\}$, but the right side is false.

$$(p \oplus q) * (p' \oplus q) \ \Rightarrow\ (p * p') \oplus q$$

Counterexample: Let $p = \mathtt{t}_1 \rightarrowtail (\gamma_1, n_1)$, $p' = \mathtt{t}_2 \rightarrowtail (\gamma_2, n_2)$ and $q = \mathtt{t}_1 \rightarrowtail (\mathbf{end}, n_1') \vee \mathtt{t}_2 \rightarrowtail (\mathbf{end}, n_2')$. Then the left side can be satisfied when $\Delta = \{(\{\mathtt{t}_1 \leadsto (\gamma_1, n_1), \mathtt{t}_2 \leadsto (\gamma_2, n_2)\}, \emptyset), (\{\mathtt{t}_1 \leadsto (\gamma_1, n_1), \mathtt{t}_2 \leadsto (\mathbf{end}, n_2')\}, \emptyset), (\{\mathtt{t}_1 \leadsto (\mathbf{end}, n_1'), \mathtt{t}_2 \leadsto (\gamma_2, n_2)\}, \emptyset), (\{\mathtt{t}_1 \leadsto (\mathbf{end}, n_1'), \mathtt{t}_2 \leadsto (\mathbf{end}, n_2')\}, \emptyset)\}$, but the right side cannot.

(4) Distributivity of $\wedge$ over $\oplus$.
$$(p \oplus p') \wedge q \ \Rightarrow\ (p \wedge q) \oplus (p' \wedge q)$$

Counterexample: Let $p = \mathtt{t} \rightarrowtail (\gamma, n)$, $p' = \mathtt{t} \rightarrowtail (\mathbf{end}, n')$ and $q = p \oplus p'$.

$$(p \wedge q) \oplus (p' \wedge q) \ \Rightarrow\ (p \oplus p') \wedge q$$

Counterexample: Let $p = p' = q = \mathtt{t} \rightarrowtail (\gamma, n) \vee \mathtt{t} \rightarrowtail (\mathbf{end}, n')$. It does not hold because $q \oplus q \Rightarrow q$ does not hold.

B.2 Actions

Properties for $*$ and $\oplus$.

(1) Commutativity.

$$R * R' \Leftrightarrow R' * R$$
$$R \oplus R' \Leftrightarrow R' \oplus R$$

(2) Associativity.

$$(R_1 * R_2) * R_3 \Leftrightarrow R_1 * (R_2 * R_3)$$
$$(R_1 \oplus R_2) \oplus R_3 \Leftrightarrow R_1 \oplus (R_2 \oplus R_3)$$

(3) Neutral element.

$$R * \mathsf{Emp} \Leftrightarrow R$$

(4) Monotonicity.

$$\frac{R_1 \Rightarrow R_1' \qquad R_2 \Rightarrow R_2'}{R_1 * R_2 \Rightarrow R_1' * R_2'}$$
$$\frac{R_1 \Rightarrow R_1' \qquad R_2 \Rightarrow R_2'}{R_1 \oplus R_2 \Rightarrow R_1' \oplus R_2'}$$

(5) Exchange laws with $\ltimes$.

$$(p * p') \ltimes (q * q') \Leftrightarrow (p \ltimes q) * (p' \ltimes q')$$
$$(p \oplus p') \ltimes (q \oplus q') \Leftrightarrow (p \ltimes q) \oplus (p' \ltimes q')$$

(6) Idempotence.

$$R \Rightarrow R \oplus R$$
$$\mathsf{True} \Leftrightarrow \mathsf{True} \oplus \mathsf{True}$$

Properties for stability and fence.

(1) Compositionality of fence.

$$\frac{I \triangleright R \qquad I' \triangleright R'}{I * I' \triangleright R * R'}$$
$$\frac{I \triangleright R \qquad I' \triangleright R' \qquad \mathsf{Precise}(I \oplus I')}{I \oplus I' \triangleright R \oplus R'}$$

A counterexample for the following:

If $\mathsf{Precise}(p)$ and $\mathsf{Precise}(q)$, then $\mathsf{Precise}(p \oplus q)$.

Let $p = (\mathsf{t}_1 \rightarrowtail (\gamma_1, n_1) * \mathsf{t}_2 \rightarrowtail (\gamma_2, n_2)) \vee \mathsf{t}_1 \rightarrowtail (\mathbf{end}, n_1')$ and $q = (\mathsf{t}_1 \rightarrowtail (\mathbf{end}, n_1') * \mathsf{t}_2 \rightarrowtail (\gamma_2, n_2)) \vee \mathsf{t}_1 \rightarrowtail (\gamma_1, n_1)$. Then $p \oplus q = ((\mathsf{t}_1 \rightarrowtail (\gamma_1, n_1) * \mathsf{t}_2 \rightarrowtail (\gamma_2, n_2)) \oplus (\mathsf{t}_1 \rightarrowtail (\mathbf{end}, n_1') * \mathsf{t}_2 \rightarrowtail (\gamma_2, n_2))) \vee (\mathsf{t}_1 \rightarrowtail (\gamma_1, n_1) \oplus \mathsf{t}_1 \rightarrowtail (\mathbf{end}, n_1'))$, which is not precise.

(2) Compositionality of stability.

$$\frac{\mathsf{Sta}(p, R) \qquad \mathsf{Sta}(p', R') \qquad p \Rightarrow I \qquad I \triangleright R}{\mathsf{Sta}(p * p', R * R')}$$

B.3 Properties for $q \nmid p$

$q \nmid p$ is defined in Figure 9, and used in the COMMIT-SPEC-CONJ rule. It has the following inference rules:

$$\frac{q_1 \nmid p \qquad q_2 \nmid p}{(q_1 \oplus q_2) \nmid p} \qquad \frac{q_1 \nmid p \qquad q_2 \nmid p}{(q_1 \vee q_2) \nmid p} \qquad \frac{q \Rightarrow q' \qquad q' \nmid p}{q \nmid p}$$

Note that if $q_1 \nmid p$ and $q_2 \nmid p$ hold, then $(q_1 * q_2) \nmid p$ does not necessarily hold.

B.4 Examples of COMMIT-SPEC-CONJ and MULTI-COMMIT rules

Example B.1 We can prove the following:

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

First, from the COMMIT rule and the FRAME rule, we know

$\{\mathsf{t}_1 \rightarrowtail (\gamma_1, n_1) * \mathsf{t}_2 \rightarrowtail (\gamma_2, n_2)\}$
$\mathbf{commit}(\mathsf{t} \rightarrowtail (\gamma_1, n_1))$
$\{\mathsf{t}_1 \rightarrowtail (\gamma_1, n_1) * \mathsf{t}_2 \rightarrowtail (\gamma_2, n_2)\}$

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

Then, by the COMMIT-SPEC-CONJ rule, we know

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

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

Finally by the MULTI-COMMIT rule, we get the conclusion:

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

Example B.2 Similarly, we can prove the following:

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

Example B.3 Why does the MULTI-COMMIT rule need the side-condition $\mathsf{Exact}(\{p_1, p_2\})$? We show the following example:

$$p \stackrel{\text{def}}{=} (y = 0) * (z = 0) * (\mathsf{t} \rightarrowtail (\gamma, n))$$
$$p_1 \stackrel{\text{def}}{=} (x = 0 \vee y = 0) * (\mathsf{t} \rightarrowtail (\gamma, n))$$
$$p_2 \stackrel{\text{def}}{=} (x = 0 \vee z = 0) * (\mathsf{t} \rightarrowtail (\gamma, n))$$

We know $\mathsf{SpecExact}(\{p_1, p_2\})$, and by applying the COMMIT and FRAME rules, we have

$$\{p\}\mathbf{commit}(p_1)\{p\}\,, \quad \{p\}\mathbf{commit}(p_2)\{p\}$$

Since $p_1 \oplus p_2 = ((x = 0) * (\mathsf{t} \rightarrowtail (\gamma, n)))$, we know it is satisfiable. However, $\mathbf{commit}(p_1 \oplus p_2)$ will abort if executed from a state satisfying p.

C. Logic Soundness Proofs

In this section, we prove our logic is sound via the simulation in Definition 7. We prove Lemmas 8 and 9 below, and get the final soundness theorem (Theorem 10) directly from them.

C.1 Proofs of Lemma 8 (Simulation Implies Contextual Refinement)

We define two auxiliary simulations:

1. A thread-local simulation between concrete and abstract client code, including method calls (Definition 21 below). We will show:
   (a) it is implied by the simulation for method in Definition 7 when the two levels are method calls (the first rule in Figure 20);
   (b) it trivially holds for client commands (the second and third lines in Figure 20);
   (c) it is also compositional (other rules in Figure 20) and could ensure the following whole program simulation.
2. A whole-program simulation between concrete and abstract levels (Definition 20 below). We will show it implies a subset relation between observable behaviors of the two levels (Lemma 35), thus could ensure contextual refinement.

Then, Lemma 8 is proved as follows:

Proof: To show $\Pi \sqsubseteq_\varphi \Gamma$, we want to prove: for any $n, C_1, \ldots, C_n$, σ_c, σ_o, and θ, if $(\sigma_o, \theta) \in \varphi$ (here we simply view φ as a relation), then

$$\begin{aligned}&\mathcal{O}[\![(\textbf{let}\ \Pi\ \textbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_o)]\!]\\ &\quad \subseteq \mathcal{O}[\![(\textbf{with}\ \Gamma\ \textbf{do}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \theta)]\!].\end{aligned}$$

By Lemma 35 (the whole-program simulation implies a behavior-subset relation), we only need to prove:

$$(\textbf{let}\ \Pi\ \textbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n) \preceq_\varphi (\textbf{with}\ \Gamma\ \textbf{do}\ C_1 \,\|\, \ldots \,\|\, C_n)$$

Also, since $\lfloor\varphi\rfloor \Rightarrow p_{\mathsf{t}}$, we know $\lfloor\varphi\rfloor \Rightarrow \lfloor p_{\mathsf{t}}\rfloor_\Gamma$.
since $R_{\mathsf{t}} = \bigvee_{\mathsf{t}'\neq\mathsf{t}} G_{\mathsf{t}'}$, we know $\lfloor R_{\mathsf{t}}\rfloor_\Gamma = \bigvee_{\mathsf{t}'\neq\mathsf{t}} \lfloor G_{\mathsf{t}'}\rfloor_\Gamma$;
since $I \triangleright \{R_{\mathsf{t}}, G_{\mathsf{t}}\}$, we know $\lfloor I\rfloor_\Gamma \triangleright \{\lfloor R_{\mathsf{t}}\rfloor_\Gamma, \lfloor G_{\mathsf{t}}\rfloor_\Gamma\}$;
since $p_{\mathsf{t}} \Rightarrow I$, we know $\lfloor p_{\mathsf{t}}\rfloor_\Gamma \Rightarrow \lfloor I\rfloor_\Gamma$;
since $\mathsf{Sta}(p_{\mathsf{t}}, R_{\mathsf{t}})$, we know $\mathsf{Sta}(\lfloor p_{\mathsf{t}}\rfloor_\Gamma, \lfloor R_{\mathsf{t}}\rfloor_\Gamma)$.
Thus, we can apply the parallel compositionality of the simulation for thread (Figure 20), then we only need to show: for any t,

$$(C_{\mathsf{t}}, \Pi) \preceq^i_{\lfloor R_{\mathsf{t}}\rfloor_\Gamma;\lfloor G_{\mathsf{t}}\rfloor_\Gamma;\lfloor p_{\mathsf{t}}\rfloor_\Gamma} (\mathbb{C}_{\mathsf{t}}, \Gamma)$$

It is proved by induction over the structure of C_{t}. For the base case, we have the first to third lines in Figure 20; for the inductive step, we have other compositionality rules in Figure 20. ☐

C.1.1 Definitions of Simulations for Thread and Program

We first define some notations in Figure 21. Most operations are simply lifted from those defined for the simulation for method (Sections 4 and 5).

Definition 20 (Simulation for Program). $W \preceq_{\widetilde{p}} \mathbb{W}$ iff

$$\begin{aligned}&\forall \sigma_c, \sigma_o, \theta, \mathcal{K}, \mathbb{K}.\ (\sigma_o, \theta) \in \widetilde{p} \wedge (\forall \mathsf{t}.\ \mathcal{K}(\mathsf{t}) = \circ) \wedge (\forall \mathsf{t}.\ \mathbb{K}(\mathsf{t}) = \circ)\\ &\Longrightarrow\ (W, (\sigma_c, \sigma_o, \mathcal{K})) \preceq \{(\mathbb{W}, (\sigma_c, \theta, \mathbb{K}))\}\,.\end{aligned}$$

Whenever $(W, \mathcal{S}) \preceq \Omega$, then

1. if $(W, \mathcal{S}) \overset{e}{\longmapsto} (W', \mathcal{S}')$,
then there exist Ω' and H such that $\Omega \overset{H}{\rightrightarrows} \Omega'$,
$\mathsf{get\_obsv}(e) = H$ and $(W', \mathcal{S}') \preceq \Omega'$;
2. if $W = \textbf{skip}$, then $(\textbf{skip}, \_) \in \Omega$;
3. if $(W, \mathcal{S}) \overset{e}{\longmapsto} \textbf{abort}$,
then there exist $\mathbb{W}$, $\mathbb{S}$ and H such that $(\mathbb{W}, \mathbb{S}) \in \Omega$,
$(\mathbb{W}, \mathbb{S}) \overset{H}{\Longrightarrow}{}^* \textbf{abort}$ and $\mathsf{get\_obsv}(e) = \mathsf{get\_obsv}(H)$.

Definition 21 (Simulation for Thread). $(C, \Pi) \preceq^{\mathsf{t}}_{\mathbb{R};\mathbb{G};\mathbb{P}} (\mathbb{C}, \Gamma)$ iff

$$\begin{aligned}&\forall \sigma_c, \sigma_o, \Lambda.\ (\sigma_o, \Lambda) \in \mathbb{P}\\ &\Longrightarrow\ (C, (\sigma_c, \sigma_o, \circ), \Pi) \preceq^{\mathsf{t}}_{\mathbb{R};\mathbb{G};\mathbb{P}} (\Lambda * \{(\{\mathsf{t} \rightsquigarrow \mathbb{C}\}, \emptyset)\}, \circ, \Gamma)\end{aligned}$$

Whenever $(C, (\sigma_c, \sigma_o, \kappa), \Pi) \preceq^{\mathsf{t}}_{\mathbb{R};\mathbb{G};\mathbb{P}} (\Lambda, \mathit{ak}, \Gamma)$, then

1. $(\kappa = (\_, x, \_)) \Rightarrow (\mathit{ak} = (x, \_))$ and $(\kappa = \circ) \Rightarrow (\mathit{ak} = \circ)$;
2. if $(C, (\sigma_c, \sigma_o, \kappa)) \overset{e}{\longrightarrow}_{\mathsf{t},\Pi} (C', (\sigma'_c, \sigma'_o, \kappa'))$,
then there exist Λ', ak' and H such that
$(\Lambda, \sigma_c, \mathit{ak}) \overset{H}{\rightrightarrows}_{\mathsf{t},\Gamma} (\Lambda', \sigma'_c, \mathit{ak}')$, $\mathsf{get\_obsv}(e) = H$,
$((\sigma_o, \Lambda), (\sigma'_o, \Lambda')) \in \mathbb{G} \uplus \mathsf{True}$ and
$(C', (\sigma'_c, \sigma'_o, \kappa'), \Pi) \preceq^{\mathsf{t}}_{\mathbb{R};\mathbb{G};\mathbb{P}} (\Lambda', \mathit{ak}', \Gamma)$;
3. if $(C, (\sigma_c, \sigma_o, \kappa)) \overset{e}{\longrightarrow}_{\mathsf{t},\Pi} \textbf{abort}$,
then $\kappa = \circ$ and there exists H such that
$(\Lambda, \sigma_c, \mathit{ak}) \overset{H}{\rightrightarrows}_{\mathsf{t},\Gamma} \textbf{abort}$ and $\mathsf{get\_obsv}(e) = H$;
4. if $C = \textbf{skip}$,
then $\kappa = \circ$ and there exists Λ' such that
$\Lambda = \Lambda' * \{(\{\mathsf{t} \rightsquigarrow \textbf{skip}\}, \emptyset)\}$ and $(\sigma_o, \Lambda') \in \mathbb{P}$;
5. for any σ'_c, σ'_o and Λ', if $((\sigma_o, \Lambda), (\sigma'_o, \Lambda')) \in \mathbb{R} \uplus \mathsf{Id}$,
then $(C, (\sigma'_c, \sigma'_o, \kappa), \Pi) \preceq^{\mathsf{t}}_{\mathbb{R};\mathbb{G};\mathbb{P}} (\Lambda', \mathit{ak}, \Gamma)$.

$$\frac{\Pi(f) \preceq^{\mathsf{t}}_{R;G;p} \Gamma(f) \qquad \Pi(f) = (x, \_) \qquad x \notin \mathit{dom}(I)}{(y := f(E), \Pi) \preceq^{\mathsf{t}}_{\lfloor R\rfloor_\Gamma;\lfloor G\rfloor_\Gamma;\lfloor p\rfloor_\Gamma} (y := f(E), \Gamma)}$$

$$\frac{}{(c, \Pi) \preceq^{\mathsf{t}}_{\lfloor R\rfloor_\Gamma;\lfloor G\rfloor_\Gamma;\lfloor p\rfloor_\Gamma} (c, \Gamma)} \qquad \frac{}{(\textbf{skip}, \Pi) \preceq^{\mathsf{t}}_{\mathbb{R};\mathbb{G};\mathbb{P}} (\textbf{skip}, \Gamma)}$$

$$\frac{}{(\langle C\rangle, \Pi) \preceq^{\mathsf{t}}_{\lfloor R\rfloor_\Gamma;\lfloor G\rfloor_\Gamma;\lfloor p\rfloor_\Gamma} (\langle C\rangle, \Gamma)}$$

$$\frac{(C_1, \Pi) \preceq^{\mathsf{t}}_{\mathbb{R};\mathbb{G};\mathbb{P}} (\mathbb{C}_1, \Gamma) \qquad (C_2, \Pi) \preceq^{\mathsf{t}}_{\mathbb{R};\mathbb{G};\mathbb{P}} (\mathbb{C}_2, \Gamma)}{(C_1; C_1, \Pi) \preceq^{\mathsf{t}}_{\mathbb{R};\mathbb{G};\mathbb{P}} (\mathbb{C}_2; \mathbb{C}_2, \Gamma)}$$

$$\frac{(C_1, \Pi) \preceq^{\mathsf{t}}_{\mathbb{R};\mathbb{G};\mathbb{P}} (\mathbb{C}_1, \Gamma) \qquad (C_2, \Pi) \preceq^{\mathsf{t}}_{\mathbb{R};\mathbb{G};\mathbb{P}} (\mathbb{C}_2, \Gamma)}{(\textbf{if}\ (B)\ C_1\ \textbf{else}\ C_2, \Pi) \preceq^{\mathsf{t}}_{\mathbb{R};\mathbb{G};\mathbb{P}} (\textbf{if}\ (B)\ \mathbb{C}_1\ \textbf{else}\ \mathbb{C}_2, \Gamma)}$$

$$\frac{(C, \Pi) \preceq^{\mathsf{t}}_{\mathbb{R};\mathbb{G};\mathbb{P}} (\mathbb{C}, \Gamma)}{(\textbf{while}\ (B)\{C\}, \Pi) \preceq^{\mathsf{t}}_{\mathbb{R};\mathbb{G};\mathbb{P}} (\textbf{while}\ (B)\{\mathbb{C}\}, \Gamma)}$$

$$\frac{\begin{array}{c}(C_i, \Pi) \preceq^{i}_{\mathbb{R}_i;\mathbb{G}_i;\mathbb{P}_i} (\mathbb{C}_i, \Gamma) \qquad \mathbb{R}_i = \bigvee_{j\neq i} \mathbb{G}_j\\ \mathbb{I} \triangleright \{\mathbb{R}_i, \mathbb{G}_i\} \qquad \mathbb{P}_i \Rightarrow \mathbb{I} \qquad \lfloor\widetilde{p}\rfloor \Rightarrow \mathbb{P}_i \qquad \forall i \in \{1, \ldots, n\}\end{array}}{\textbf{let}\ \Pi\ \textbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n \preceq_{\widetilde{p}} \textbf{with}\ \Gamma\ \textbf{do}\ \mathbb{C}_1 \,\|\, \ldots \,\|\, \mathbb{C}_n}$$

Auxiliary definitions:

$$\lfloor\Upsilon\rfloor_\Gamma \overset{\text{def}}{=} \begin{cases}\textbf{fexec}(f, n) & \text{if } \Upsilon = (\gamma, n) \text{ and } \Gamma(f) = \gamma\\ \textbf{fret}(n') & \text{if } \Upsilon = (\textbf{end}, n')\end{cases}$$

$$\lfloor U\rfloor_\Gamma(\mathsf{t}) \overset{\text{def}}{=} \begin{cases}\lfloor\Upsilon\rfloor_\Gamma & \text{if } U(\mathsf{t}) = \Upsilon\\ \textit{undefined} & \text{otherwise}\end{cases}$$

$$\lfloor\Delta\rfloor_\Gamma \overset{\text{def}}{=} \begin{cases}\emptyset & \text{if } \Delta = \emptyset\\ \{(\lfloor U\rfloor_\Gamma, \theta)\} \uplus \lfloor\Delta'\rfloor_\Gamma & \text{if } \Delta = \{(U, \theta)\} \uplus \Delta'\end{cases}$$

$(\sigma, \lfloor\Delta\rfloor_\Gamma) \in \lfloor p\rfloor_\Gamma$ iff $(\sigma, \Delta) \models p$

$((\sigma, \lfloor\Delta\rfloor_\Gamma), (\sigma', \lfloor\Delta'\rfloor_\Gamma)) \in \lfloor R\rfloor_\Gamma$ iff $((\sigma, \Delta), (\sigma', \Delta')) \models R$

$\mathsf{fstep}(\mathbb{R}) \overset{\text{def}}{=} \exists R, \Gamma.\ \mathbb{R} = \lfloor R\rfloor_\Gamma$

$x \notin \mathit{dom}(I) \overset{\text{def}}{=} \forall \sigma, \Delta.\ ((\sigma, \Delta) \models I) \Longrightarrow x \notin \mathit{dom}(\sigma)$

Figure 20. Compositionality Rules for Simulation

We can prove the following lemma about $\rightrightarrows$:

Lemma 22. If $\Delta \rightrightarrows \Delta'$, then $(\lfloor\Delta\rfloor_\Gamma, \sigma_c, \mathit{ak}) \rightrightarrows_{\mathsf{t},\Gamma} (\lfloor\Delta'\rfloor_\Gamma, \sigma_c, \mathit{ak})$ holds for any σ_c, ak and t.

C.1.2 Simulation for Thread is Lifted from Simulation for Method, and is Compositional

We show the compositionality rules in Figure 20. Here we assume there exists $\mathbb{I}$ such that at each rule, $\mathbb{I} \triangleright \{\mathbb{R}, \mathbb{G}\}$, $\mathbb{P} \Rightarrow \mathbb{I}$, $\mathsf{Sta}(\mathbb{P}, \mathbb{R})$ and $\mathsf{fstep}(\{\mathbb{R}, \mathbb{G}\})$ hold. We prove their soundness in the following Lemmas 23, 25, 26, 27, 28, 31, 32 and 33.

Lemma 23 (Sim for Thread is Lifted from Sim for Method). If $\Pi(f) \preceq^{\mathsf{t}}_{R;G;p} \Gamma(f)$,
$\Pi(f) = (x, C)$, $x \notin \mathit{dom}(I)$, $I \triangleright \{R, G\}$ and $p \Rightarrow I$,
then $(y := f(E), \Pi) \preceq^{\mathsf{t}}_{\lfloor R\rfloor_\Gamma;\lfloor G\rfloor_\Gamma;\lfloor p\rfloor_\Gamma} (y := f(E), \Gamma)$.

Proof: Suppose $\Gamma(f) = \gamma$. The premise tells us:

$$\begin{aligned}&\forall n, \sigma, \Delta.\ (\sigma, \Delta) \models (\mathsf{t} \rightarrowtail (\gamma, n)) * (x = n) * p)\\ &\quad\Longrightarrow\ (C; \textbf{noret}, \sigma) \preceq^{\mathsf{t}}_{R;G;p} \Delta\,.\end{aligned}$$

We want to prove:

$$\begin{aligned}&\forall \sigma_c, \sigma_o, \Lambda.\ (\sigma_o, \Lambda) \in \lfloor p\rfloor_\Gamma\\ &\Longrightarrow\ (y := f(E), (\sigma_c, \sigma_o, \circ), \Pi) \preceq^{\mathsf{t}}_{\lfloor R\rfloor_\Gamma;\lfloor G\rfloor_\Gamma;\lfloor p\rfloor_\Gamma}\\ &\qquad (\Lambda * \{(\{\mathsf{t} \rightsquigarrow (y := f(E))\}, \emptyset)\}, \circ, \Gamma)\end{aligned}$$

$$
\begin{array}{l}
\Omega \ ::= \ \{(\mathbb{W},\mathbb{S})\}^* \\
\widetilde{p} \ ::= \ \{(\sigma_o,\theta)\}^* \\
\mathbb{U} \ \in \ \mathit{ThrdID} \rightharpoonup \mathit{AbsStmt} \\
\Lambda \ ::= \ \{(\mathbb{U},\theta)\}^* \\
\mathbb{P},\mathbb{I} \ ::= \ \{(\sigma_o,\Lambda)\}^* \\
\mathbb{R},\mathbb{G} \ ::= \ \{((\sigma_o,\Lambda),(\sigma_o',\Lambda'))\}^*
\end{array}
$$

$$
\begin{array}{l}
\Omega \overset{H}{\rightrightarrows} \Omega' \ \text{iff} \\
\quad \forall \mathbb{W}',\mathbb{S}'.\, (\mathbb{W}',\mathbb{S}') \in \Omega' \\
\quad \Longrightarrow \ \exists \mathbb{W},\mathbb{S},H'.\, (\mathbb{W},\mathbb{S}) \in \Omega \wedge (\mathbb{W},\mathbb{S}) \overset{H'}{\phi\!\!\longrightarrow}{}^* (\mathbb{W}',\mathbb{S}') \\
\qquad\qquad\qquad \wedge\ \mathsf{get\_obsv}(H') = H
\end{array}
$$

$$
\frac{\mathbb{U}(\mathsf{t}) = \mathbb{C} \qquad (\mathbb{C},(\sigma_c,\theta,\kappa)) \overset{H}{\circ\!\!\longrightarrow}_{\mathsf{t},\Gamma} (\mathbb{C}',(\sigma_c',\theta',\kappa'))}{(\mathbb{U},(\sigma_c,\theta,\mathit{ak})) \overset{H}{\dashrightarrow}_{\mathsf{t},\Gamma} (\mathbb{U}\{\mathsf{t} \leadsto \mathbb{C}'\},(\sigma_c',\theta',\mathit{ak}'))}
$$

$$
\frac{\mathsf{t}' \neq \mathsf{t} \qquad \mathbb{U}(\mathsf{t}') = \mathbf{fexec}(f,n) \qquad \Gamma(f)(n)(\theta) = (n',\theta')}{(\mathbb{U},(\sigma_c,\theta,\mathit{ak})) \dashrightarrow_{\mathsf{t},\Gamma} (\mathbb{U}\{\mathsf{t}' \leadsto \mathbf{fret}(n')\},(\sigma_c,\theta',\mathit{ak}))}
$$

$$
\begin{array}{l}
(\Lambda,\sigma_c,\mathit{ak}) \overset{H}{\rightrightarrows}_{\mathsf{t},\Gamma} (\Lambda',\sigma_c',\mathit{ak}') \ \text{iff} \\
\quad \forall \mathbb{U}',\theta'.\, (\mathbb{U}',\theta') \in \Lambda' \\
\quad \Longrightarrow \ \exists \mathbb{U},\theta,H'.\, (\mathbb{U},\theta) \in \Lambda \\
\qquad\qquad \wedge\ (\mathbb{U},(\sigma_c,\theta,\mathit{ak})) \overset{H'}{\dashrightarrow}{}^*_{\mathsf{t},\Gamma} (\mathbb{U}',(\sigma_c',\theta',\mathit{ak}')) \\
\qquad\qquad \wedge\ \mathsf{get\_obsv}(H') = H
\end{array}
$$

$$
\begin{array}{l}
(\Lambda,\sigma_c,\mathit{ak}) \overset{H}{\rightrightarrows}_{\mathsf{t},\Gamma} \mathbf{abort} \ \text{iff} \\
\quad \exists \Lambda',\sigma_c',\mathit{ak}',H'.\, (\Lambda,\sigma_c,\mathit{ak}) \overset{H'}{\rightrightarrows}_{\mathsf{t},\Gamma} (\Lambda',\sigma_c',\mathit{ak}') \\
\quad\ \wedge\ \exists \mathbb{U},\theta,H''.\, (\mathbb{U},\theta) \in \Lambda' \wedge (\mathbb{U}(\mathsf{t}),(\sigma_c',\theta,\mathit{ak}')) \overset{H''}{\circ\!\!\longrightarrow}{}^*_{\mathsf{t},\Gamma} \mathbf{abort} \\
\quad\ \wedge\ \mathsf{get\_obsv}(H' :: H'') = H
\end{array}
$$

$$
\begin{array}{l}
\Lambda * \Lambda' \ \overset{\text{def}}{=} \ \{(\mathbb{U} \uplus \mathbb{U}',\theta \uplus \theta') \mid (\mathbb{U},\theta) \in \Lambda \wedge (\mathbb{U}',\theta') \in \Lambda'\} \\
\mathbb{P} \uplus \mathbb{P}' \ \overset{\text{def}}{=} \ \{(\sigma \uplus \sigma',\Lambda * \Lambda') \mid (\sigma,\Lambda) \in \mathbb{P} \wedge (\sigma',\Lambda') \in \mathbb{P}'\} \\
\mathbb{R} \uplus \mathbb{R}' \ \overset{\text{def}}{=} \ \{(\sigma_1 \uplus \sigma_1',\Lambda_1 * \Lambda_1'),(\sigma_2 \uplus \sigma_2',\Lambda_2 * \Lambda_2') \\
\qquad\quad \mid ((\sigma_1,\Lambda_1),(\sigma_2,\Lambda_2)) \in \mathbb{R} \wedge ((\sigma_1',\Lambda_1'),(\sigma_2',\Lambda_2')) \in \mathbb{R}'\}
\end{array}
$$

Figure 21. Auxiliary Definitions for Simulation

For any σ_c, σ_o and Λ, if $(\sigma_o,\Lambda) \in \lfloor p \rfloor_\Gamma$, then there exists Δ such that $(\sigma_o,\Delta) \models p$ and $\Lambda = \lfloor \Delta \rfloor_\Gamma$. Thus for any n,

$$
\begin{array}{l}
(\sigma_o \uplus \{x \leadsto n\}, \Delta * \{(\{\mathsf{t} \leadsto (\gamma,n)\},\emptyset)\}) \models \\
(\mathsf{t} \rightarrowtail (\gamma,n) * (x = n) * p)\,.
\end{array}
$$

From the premise, we know

$$(C;\mathbf{noret},\sigma_o \uplus \{x \leadsto n\}) \preceq^{\mathsf{t}}_{R;G;p} \Delta * \{(\{\mathsf{t} \leadsto (\gamma,n)\},\emptyset)\}\,.$$

Since

$$\lfloor \Delta * \{(\{\mathsf{t} \leadsto (\gamma,n)\},\emptyset)\} \rfloor_\Gamma = \Lambda * \{(\{\mathsf{t} \leadsto \mathbf{fexec}(f,n)\},\emptyset)\}$$

from the following Lemma 24, we have

$$
\begin{array}{l}
(C;\mathbf{noret},(\sigma_c,\sigma_o,\kappa),\Pi) \preceq^{\mathsf{t}}_{\lfloor R \rfloor_\Gamma;\lfloor G \rfloor_\Gamma;\lfloor p \rfloor_\Gamma} \\
(\Lambda * \{(\{\mathsf{t} \leadsto \mathbf{fexec}(f,n)\},\emptyset)\},\mathit{ak},\Gamma)
\end{array}
$$

where $\kappa = (\{x \leadsto n\},y,\mathbf{skip})$ and $\mathit{ak} = (y,\mathbf{skip})$.
Then we can prove

$$
\begin{array}{l}
(y := f(E),(\sigma_c,\sigma_o,\circ),\Pi) \preceq^{\mathsf{t}}_{\lfloor R \rfloor_\Gamma;\lfloor G \rfloor_\Gamma;\lfloor p \rfloor_\Gamma} \\
(\Lambda * \{(\{\mathsf{t} \leadsto (y := f(E))\},\emptyset)\},\circ,\Gamma)
\end{array}
$$

by definition and operational semantics. □

Lemma 24. For any C, σ_o, σ_l, κ, x, y, Δ, ak, R, G and p, if

1. $(C,\sigma_o \uplus \sigma_l) \preceq^{\mathsf{t}}_{R;G;p} \Delta$,
2. $\exists C'.\, C = (C';\mathbf{noret})$, and $\sigma_l = \{x \leadsto \_\}$,
3. there exists I such that $I \triangleright \{R,G\}$, $(\sigma_o,\Delta) \models I * \mathsf{true}$, $p \Rightarrow I$, $x \notin \mathit{dom}(I)$,
4. $\kappa = (\sigma_l,y,\mathbf{skip})$ and $\mathit{ak} = (y,\mathbf{skip})$,

then for any σ_c, $(C,(\sigma_c,\sigma_o,\kappa),\Pi) \preceq^{\mathsf{t}}_{\lfloor R \rfloor_\Gamma;\lfloor G \rfloor_\Gamma;\lfloor p \rfloor_\Gamma} (\lfloor \Delta \rfloor_\Gamma,\mathit{ak},\Gamma)$.

Proof: By definition and co-induction. We have the following cases:

1. If $(C,(\sigma_c,\sigma_o,\kappa)) \xrightarrow{e}_{\mathsf{t},\Pi} (C',(\sigma_c',\sigma_o',\circ))$,
   by the operational semantics, we know $C' = \mathbf{skip}$, $\sigma_o' = \sigma_o$, $C = \mathbf{E}[\,\mathbf{return}\ E\,]$, $\mathsf{get\_obsv}(e) = \epsilon$, and there exists n such that $[\![E]\!]_{\sigma_o \uplus \sigma_l} = n$ and $\sigma_c' = \sigma_c\{y \leadsto n\}$.
   From the first premise, we know
   $$(\sigma_o \uplus \sigma_l,\Delta) \models (\mathsf{t} \rightarrowtail (\mathbf{end},n) * (x = \_) * p).$$
   Thus there exists Δ' such that $(\sigma_o,\Delta') \models p$ and
   $$\Delta = \Delta' * \{(\{\mathsf{t} \leadsto (\mathbf{end},n)\},\emptyset)\}\,.$$
   Thus
   $$\lfloor \Delta \rfloor_\Gamma = \lfloor \Delta' \rfloor_\Gamma * \{(\{\mathsf{t} \leadsto \mathbf{fret}(n)\},\emptyset)\}\,.$$
   By the abstract operational semantics, we know: for any θ,
   $$(\mathbf{fret}(n),(\sigma_c,\theta,\mathit{ak})) \circ\!\!\longrightarrow_{\mathsf{t},\Gamma} (\mathbf{skip},(\sigma_c',\theta,\circ))$$
   Thus we have:
   $$(\lfloor \Delta \rfloor_\Gamma,\sigma_c,\mathit{ak}) \rightrightarrows_{\mathsf{t},\Gamma} (\lfloor \Delta' \rfloor_\Gamma * \{(\{\mathsf{t} \leadsto \mathbf{skip}\},\emptyset)\},\sigma_c',\circ)\,.$$
   Since $(\sigma_o,\Delta') \models p$ and $p \Rightarrow I$, we know $(\sigma_o,\Delta') \models I$. Thus $(\sigma_o,\lfloor \Delta' \rfloor_\Gamma) \models \lfloor I \rfloor_\Gamma$. Since $I \triangleright G$, we know
   $$
   \begin{array}{l}
   ((\sigma_o,\lfloor \Delta \rfloor_\Gamma),(\sigma_o,\lfloor \Delta' \rfloor_\Gamma * \{(\{\mathsf{t} \leadsto \mathbf{skip}\},\emptyset)\})) \\
   \in \lfloor G \rfloor_\Gamma \uplus \mathsf{True}.
   \end{array}
   $$
   Since $(\sigma_o,\lfloor \Delta' \rfloor_\Gamma) \models \lfloor p \rfloor_\Gamma$, by Lemma 26, we have
   $$
   \begin{array}{l}
   (\mathbf{skip},(\sigma_c',\sigma_o,\circ),\Pi) \preceq^{\mathsf{t}}_{\lfloor R \rfloor_\Gamma;\lfloor G \rfloor_\Gamma;\lfloor p \rfloor_\Gamma} \\
   (\lfloor \Delta' \rfloor_\Gamma * \{(\{\mathsf{t} \leadsto \mathbf{skip}\},\emptyset)\},\circ,\Gamma).
   \end{array}
   $$
2. If $(C,(\sigma_c,\sigma_o,\kappa)) \xrightarrow{e}_{\mathsf{t},\Pi} (C',(\sigma_c',\sigma_o',\kappa'))$ and there exists σ_l' such that $\kappa' = (\sigma_l',y,\mathbf{skip})$,
   by the operational semantics, we know $\sigma_c' = \sigma_c$,
   $$(C,\sigma_o \uplus \sigma_l) \longrightarrow_{\mathsf{t}} (C',\sigma' \uplus \sigma_l'),$$
   $\mathit{dom}(\sigma_l) = \mathit{dom}(\sigma_l')$ and $\mathsf{get\_obsv}(e) = \epsilon$.
   From the first premise, we know there exists Δ' such that
   $$
   \begin{array}{l}
   \Delta \rightrightarrows \Delta', \\
   ((\sigma_o \uplus \sigma_l,\Delta),(\sigma_o' \uplus \sigma_l',\Delta')) \models (G * \mathsf{True}), \\
   (C',\sigma_o' \uplus \sigma_l') \preceq^{\mathsf{t}}_{R;G;p} \Delta'.
   \end{array}
   $$
   By Lemma 22, we know
   $$(\lfloor \Delta \rfloor_\Gamma,\sigma_c,\mathit{ak}) \rightrightarrows_{\mathsf{t},\Gamma} (\lfloor \Delta' \rfloor_\Gamma,\sigma_c,\mathit{ak}).$$
   Also we know
   $$((\sigma_o \uplus \sigma_l,\lfloor \Delta \rfloor_\Gamma),(\sigma_o' \uplus \sigma_l',\lfloor \Delta' \rfloor_\Gamma)) \in \lfloor G \rfloor_\Gamma \uplus \mathsf{True}.$$
   Since $(\sigma_o,\Delta) \models I * \mathsf{true}$, $\sigma_l = \{x \leadsto \_\}$, $x \notin \mathit{dom}(I)$ and $I \triangleright G$, we know
   $$((\sigma_o,\lfloor \Delta \rfloor_\Gamma),(\sigma_o',\lfloor \Delta' \rfloor_\Gamma)) \in \lfloor G \rfloor_\Gamma \uplus \mathsf{True}.$$
   Also we have $(\sigma_o',\Delta') \models I * \mathsf{true}$. Then from the hypothesis, we have
   $$(C',(\sigma_c,\sigma_o',\kappa'),\Pi) \preceq^{\mathsf{t}}_{\lfloor R \rfloor_\Gamma;\lfloor G \rfloor_\Gamma;\lfloor p \rfloor_\Gamma} (\lfloor \Delta' \rfloor_\Gamma,\mathit{ak},\Gamma).$$

3. If $(C, (\sigma_c, \sigma_o, \kappa)) \xrightarrow{e}_{\mathsf{t},\Pi}$ **abort**,
by the operational semantics, we know
$$(C, \sigma_o \uplus \sigma_l) \longrightarrow_{\mathsf{t}} \textbf{abort},$$
which contradicts the first premise.
4. For any σ_c', σ_o' and Λ', if $((\sigma_o, \lfloor\Delta\rfloor_\Gamma), (\sigma_o', \Lambda')) \in \lfloor R\rfloor_\Gamma \uplus \mathsf{Id}$, we know there exists Δ' such that $\Lambda' = \lfloor\Delta'\rfloor_\Gamma$. Thus
$$((\sigma_o \uplus \sigma_l, \Delta), (\sigma_o' \uplus \sigma_l, \Delta')) \models (R * \mathsf{Id})$$
From the first premise, we know
$$(C, \sigma_o' \uplus \sigma_l) \preceq^{\mathsf{t}}_{R;G;p} \Delta'$$
Since $I \rhd R$ and $x \notin \mathit{dom}(I)$, we know $(\sigma_o', \Delta') \models I * \mathsf{true}$. From the hypothesis, we get
$$(C, (\sigma_c', \sigma_o', \kappa), \Pi) \preceq^{\mathsf{t}}_{\lfloor R\rfloor_\Gamma;\lfloor G\rfloor_\Gamma;\lfloor p\rfloor_\Gamma} (\lfloor\Delta'\rfloor_\Gamma, ak, \Gamma).$$

By definition, we complete the proof. □

Lemma 25 (Simulation for Thread Holds on Instruction). For any instruction c, if there exists I such that $I \rhd \{R, G\}$, $p \Rightarrow I$ and $\mathsf{Sta}(p, R)$, then $(c, \Pi) \preceq^{\mathsf{t}}_{\lfloor R\rfloor_\Gamma;\lfloor G\rfloor_\Gamma;\lfloor p\rfloor_\Gamma} (c, \Gamma)$.

Proof: We want to prove: for any σ_o and Λ, if $(\sigma_o, \Lambda) \in \lfloor p\rfloor_\Gamma$, then for any σ_c,
$$(c, (\sigma_c, \sigma_o, \circ), \Pi) \preceq^{\mathsf{t}}_{\lfloor R\rfloor_\Gamma;\lfloor G\rfloor_\Gamma;\lfloor p\rfloor_\Gamma} (\Lambda * \{(\{\mathsf{t} \rightsquigarrow c\}, \emptyset)\}, \circ, \Gamma)$$
We prove it by definition and co-induction. We have the following cases:

1. If $(c, (\sigma_c, \sigma_o, \circ)) \xrightarrow{e}_{\mathsf{t},\Pi} (\textbf{skip}, (\sigma_c', \sigma_o, \circ))$,
by the operational semantics, we know for any θ,
$$(c, (\sigma_c, \theta, \circ)) \circ\!\!\xrightarrow{e}_{\mathsf{t},\Gamma} (\textbf{skip}, (\sigma_c', \theta, \circ))$$
Thus we have
$$\begin{array}{l}(\Lambda * \{(\{\mathsf{t} \rightsquigarrow c\}, \emptyset)\}, \sigma_c, \circ) \overset{H}{\rightrightarrows}_{\mathsf{t},\Gamma} \\ (\Lambda * \{(\{\mathsf{t} \rightsquigarrow \textbf{skip}\}, \emptyset)\}, \sigma_c', \circ)\end{array}$$
where $H = \mathsf{get\_obsv}(e)$.
Also, since $p \Rightarrow I$, we know $(\sigma_o, \Lambda) \in \lfloor I\rfloor_\Gamma$. Thus
$$((\sigma_o, \Lambda), (\sigma_o, \Lambda)) \in \lfloor [I] \rfloor_\Gamma \subseteq \lfloor G\rfloor_\Gamma$$
Thus we have
$$\begin{array}{l}((\sigma_o, \Lambda * \{(\{\mathsf{t} \rightsquigarrow c\}, \emptyset)\}), (\sigma_o, \Lambda * \{(\{\mathsf{t} \rightsquigarrow \textbf{skip}\}, \emptyset)\})) \\ \in \lfloor G\rfloor_\Gamma \uplus \mathsf{True}\end{array}$$
Since $(\sigma_o, \Lambda) \in \lfloor p\rfloor_\Gamma$, by Lemma 26, we have
$$\begin{array}{l}(\textbf{skip}, (\sigma_c', \sigma_o, \circ), \Pi) \preceq^{\mathsf{t}}_{\lfloor R\rfloor_\Gamma;\lfloor G\rfloor_\Gamma;\lfloor p\rfloor_\Gamma} \\ (\Lambda * \{(\{\mathsf{t} \rightsquigarrow \textbf{skip}\}, \emptyset)\}, \circ, \Gamma).\end{array}$$
2. If $(c, (\sigma_c, \sigma_o, \circ)) \xrightarrow{e}_{\mathsf{t},\Pi}$ **abort**,
by the operational semantics, we know for any θ,
$$(c, (\sigma_c, \theta, \circ)) \circ\!\!\xrightarrow{e}_{\mathsf{t},\Gamma} \textbf{abort}$$
Thus we have
$$(\Lambda * \{(\{\mathsf{t} \rightsquigarrow c\}, \emptyset)\}, \sigma_c, \circ) \overset{H}{\rightrightarrows}_{\mathsf{t},\Gamma} \textbf{abort}$$
where $H = \mathsf{get\_obsv}(e)$.
3. For any σ_c', σ_o' and Λ', if
$$((\sigma_o, \Lambda * \{(\{\mathsf{t} \rightsquigarrow c\}, \emptyset)\}), (\sigma_o', \Lambda')) \in \lfloor R\rfloor_\Gamma \uplus \mathsf{Id},$$
since $(\sigma_o, \Lambda) \in \lfloor I\rfloor_\Gamma$ and $I \rhd R$, we know there exists Λ'' such that
$$\begin{array}{l}\Lambda' = \Lambda'' * \{(\{\mathsf{t} \rightsquigarrow c\}, \emptyset)\}) \ , \\ ((\sigma_o, \Lambda), (\sigma_o', \Lambda'')) \in \lfloor R\rfloor_\Gamma \ .\end{array}$$
Since $\mathsf{Sta}(p, R)$, we know
$$(\sigma_o', \Lambda'') \in \lfloor p\rfloor_\Gamma \ .$$
From the hypothesis, we get
$$(c, (\sigma_c', \sigma_o', \circ), \Pi) \preceq^{\mathsf{t}}_{\lfloor R\rfloor_\Gamma;\lfloor G\rfloor_\Gamma;\lfloor p\rfloor_\Gamma} (\Lambda', \circ, \Gamma) \ .$$

By definition, we complete the proof. □

Lemma 26 (Simulation for Thread Holds on Skip). If there exists $\mathbb{I}$ such that $\mathbb{I} \rhd \mathbb{R}$, $\mathbb{P} \Rightarrow \mathbb{I}$, $\mathsf{Sta}(\mathbb{P}, \mathbb{R})$ and $\mathsf{fstep}(\mathbb{R})$, then $(\textbf{skip}, \Pi) \preceq^{\mathsf{t}}_{\mathbb{R};\mathbb{G};\mathbb{P}} (\textbf{skip}, \Gamma)$.

Proof: We want to prove: for any σ_o and Λ, if $(\sigma_o, \Lambda) \in \mathbb{P}$, then for any σ_c,
$$(\textbf{skip}, (\sigma_c, \sigma_o, \circ), \Pi) \preceq^{\mathsf{t}}_{\mathbb{R};\mathbb{G};\mathbb{P}} (\Lambda * \{(\{\mathsf{t} \rightsquigarrow \textbf{skip}\}, \emptyset)\}, \circ, \Gamma)$$
We prove it by definition and co-induction. We have the following cases:

1. The **skip** case trivially holds.
2. For any σ_c', σ_o' and Λ', if
$$((\sigma_o, \Lambda * \{(\{\mathsf{t} \rightsquigarrow \textbf{skip}\}, \emptyset)\}), (\sigma_o', \Lambda')) \in \mathbb{R} \uplus \mathsf{Id},$$
since $\mathsf{fstep}(\mathbb{R})$, $(\sigma_o, \Lambda) \in \mathbb{I}$ and $\mathbb{I} \rhd \mathbb{R}$, we know there exists Λ'' such that
$$\begin{array}{l}\Lambda' = \Lambda'' * \{(\{\mathsf{t} \rightsquigarrow \textbf{skip}\}, \emptyset)\}) \ , \\ ((\sigma_o, \Lambda), (\sigma_o', \Lambda'')) \in \mathbb{R} \ .\end{array}$$
Since $\mathsf{Sta}(\mathbb{P}, \mathbb{R})$, we know
$$(\sigma_o', \Lambda'') \in \mathbb{P} \ .$$
From the hypothesis, we get
$$(\textbf{skip}, (\sigma_c', \sigma_o', \circ), \Pi) \preceq^{\mathsf{t}}_{\mathbb{R};\mathbb{G};\mathbb{P}} (\Lambda', \circ, \Gamma) \ .$$

By definition, we complete the proof. □

Lemma 27 (Simulation for Thread Holds on Atomic Block). For any client code C, if there exists I such that $I \rhd \{R, G\}$, $p \Rightarrow I$ and $\mathsf{Sta}(p, R)$, then $(\langle C\rangle, \Pi) \preceq^{\mathsf{t}}_{\lfloor R\rfloor_\Gamma;\lfloor G\rfloor_\Gamma;\lfloor p\rfloor_\Gamma} (\langle C\rangle, \Gamma)$. [1]

Proof: The proof is similar to the proof of Lemma 25. We can just view $\langle C\rangle$ as a single-step instruction from clients. It does not access object states. □

To prove sequential compositionality below, we first need to define some useful notations:

$$\begin{array}{l}\Lambda \lhd \{\mathsf{t} \rightsquigarrow \mathbb{C}_2\} \overset{\text{def}}{=} \\ \{(\mathbb{U}\{\mathsf{t} \rightsquigarrow \mathbb{C}_1; \mathbb{C}_2\}, \theta) \mid (\mathbb{U}, \theta) \in \Lambda \wedge \mathbb{U}(\mathsf{t}) = \mathbb{C}_1\}\end{array} \quad \text{(C.1)}$$

$$\begin{array}{l}\mathsf{outf}(\Lambda, \mathsf{t}) \overset{\text{def}}{=} \\ \forall \mathbb{U}, \theta.\ (\mathbb{U}, \theta) \in \Lambda \Longrightarrow \\ \exists \mathbb{C}.\ \mathbb{U}(\mathsf{t}) = \mathbb{C} \wedge \mathbb{C} \neq \textbf{fexec}(\_, \_) \wedge \mathbb{C} \neq \textbf{fret}(\_)\end{array} \quad \text{(C.2)}$$

Lemma 28 (Sequential Compositionality of Simulation). If $(C_1, \Pi) \preceq^{\mathsf{t}}_{\mathbb{R};\mathbb{G};\mathbb{P}} (\mathbb{C}_1, \Gamma)$ and $(C_2, \Pi) \preceq^{\mathsf{t}}_{\mathbb{R};\mathbb{G};\mathbb{P}} (\mathbb{C}_2, \Gamma)$, then $(C_1; C_1, \Pi) \preceq^{\mathsf{t}}_{\mathbb{R};\mathbb{G};\mathbb{P}} (\mathbb{C}_2; \mathbb{C}_2, \Gamma)$. [2]

Proof: From the premise, we know: for any σ_c, σ_o and Λ, if $(\sigma_o, \Lambda) \in \mathbb{P}$, then
$$(C_1, (\sigma_c, \sigma_o, \circ), \Pi) \preceq^{\mathsf{t}}_{\mathbb{R};\mathbb{G};\mathbb{P}} (\Lambda * \{(\{\mathsf{t} \rightsquigarrow \mathbb{C}_1\}, \emptyset)\}, \circ, \Gamma)$$
By the following Lemma 29, we know

[1] Remember in our simple setting, $\langle C\rangle$ does not contain method calls or nested atomic blocks. But it is not difficult to support them.

[2] Here C_1, $\mathbb{C}_1$, C_2 and $\mathbb{C}_2$ are not inside method body (but they could be method calls).

$$(C_1; C_2, (\sigma_c, \sigma_o, \circ), \Pi) \preceq^{\mathsf{t}}_{\mathbb{R};\mathbb{G};\mathbb{P}} (\Lambda * \{(\{\mathsf{t} \leadsto \mathbb{C}_1\}, \emptyset)\} \lhd \{\mathsf{t} \leadsto \mathbb{C}_2\}, \circ, \Gamma)$$

where

$$\Lambda * \{(\{\mathsf{t} \leadsto \mathbb{C}_1\}, \emptyset)\} \lhd \{\mathsf{t} \leadsto \mathbb{C}_2\} = \Lambda * \{(\{\mathsf{t} \leadsto \mathbb{C}_1; \mathbb{C}_2\}, \emptyset)\}$$

Thus we get $(C_1; C_1, \Pi) \preceq^{\mathsf{t}}_{\mathbb{R};\mathbb{G};\mathbb{P}} (\mathbb{C}_2; \mathbb{C}_2, \Gamma)$. □

Lemma 29. If

1. $(C_1, (\sigma_c, \sigma_o, \circ), \Pi) \preceq^{\mathsf{t}}_{\mathbb{R};\mathbb{G};\mathbb{P}} (\Lambda, \circ, \Gamma)$,
2. $(C_2, \Pi) \preceq^{\mathsf{t}}_{\mathbb{R};\mathbb{G};\mathbb{P}} (\mathbb{C}_2, \Gamma)$,
3. $\mathsf{outf}(\Lambda, \mathsf{t})$,

then $(C_1; C_2, (\sigma_c, \sigma_o, \circ), \Pi) \preceq^{\mathsf{t}}_{\mathbb{R};\mathbb{G};\mathbb{P}} (\Lambda \lhd \{\mathsf{t} \leadsto \mathbb{C}_2\}, \circ, \Gamma)$.

Proof: By definition and co-induction. We have the following cases:

1. If $C_1 = \textbf{skip}$ and $(C_1; C_2, (\sigma_c, \sigma_o, \circ)) \longrightarrow_{\mathsf{t},\Pi} (C_2, (\sigma_c, \sigma_o, \circ))$, from the first premise, we know there exists Λ' such that
$$\Lambda = \Lambda' * \{(\{\mathsf{t} \leadsto \textbf{skip}\}, \emptyset)\} \text{ and } (\sigma_o, \Lambda') \in \mathbb{P}.$$
Thus
$$\Lambda \lhd \{\mathsf{t} \leadsto \mathbb{C}_2\} = \Lambda' * \{(\{\mathsf{t} \leadsto \textbf{skip}; \mathbb{C}_2\}, \emptyset)\}.$$
We have:
$$(\Lambda' * \{(\{\mathsf{t} \leadsto \textbf{skip}; \mathbb{C}_2\}, \emptyset)\}, \sigma_c, \circ) \rightrightarrows_{\mathsf{t},\Gamma} (\Lambda' * \{(\{\mathsf{t} \leadsto \mathbb{C}_2\}, \emptyset)\}, \sigma_c, \circ)$$
From the second premise, we know
$$(C_2, (\sigma_c, \sigma_o, \circ), \Pi) \preceq^{\mathsf{t}}_{\mathbb{R};\mathbb{G};\mathbb{P}} (\Lambda' * \{(\{\mathsf{t} \leadsto \mathbb{C}_2\}, \emptyset)\}, \circ, \Gamma).$$
Since $\mathbb{I} \rhd \mathbb{G}$ and $\mathbb{P} \Rightarrow \mathbb{I}$, we know
$$((\sigma_o, \Lambda'), (\sigma_o, \Lambda')) \in \mathbb{G}\ ,$$
thus
$$((\sigma_o, \Lambda \lhd \{\mathsf{t} \leadsto \mathbb{C}_2\}, (\sigma_o, \Lambda' * \{(\{\mathsf{t} \leadsto \mathbb{C}_2\}, \emptyset)\})) \in \mathbb{G} \uplus \mathsf{True}\ .$$

2. If $(C_1; C_2, (\sigma_c, \sigma_o, \circ)) \xrightarrow{e}_{\mathsf{t},\Pi} (C_1'; C_2, (\sigma_c', \sigma_o', \circ))$, thus $(C_1, (\sigma_c, \sigma_o, \circ)) \xrightarrow{e}_{\mathsf{t},\Pi} (C_1', (\sigma_c', \sigma_o', \circ))$.
From the first premise, we know there exist Λ', ak' and H such that
$$(\Lambda, \sigma_c, \circ) \overset{H}{\rightrightarrows}_{\mathsf{t},\Gamma} (\Lambda', \sigma_c', ak') \tag{C.3}$$
$$\mathsf{get\_obsv}(e) = H \tag{C.4}$$
$$((\sigma_o, \Lambda), (\sigma_o', \Lambda')) \in \mathbb{G} \uplus \mathsf{True} \tag{C.5}$$
$$(C_1', (\sigma_c', \sigma_o', \circ), \Pi) \preceq^{\mathsf{t}}_{\mathbb{R};\mathbb{G};\mathbb{P}} (\Lambda', ak', \Gamma) \tag{C.6}$$
From (C.6), we know $ak' = \circ$. Then from (C.3), we know $\mathsf{outf}(\Lambda', \mathsf{t})$.
From the hypothesis, we have
$$(C_1'; C_2, (\sigma_c', \sigma_o', \circ), \Pi) \preceq^{\mathsf{t}}_{\mathbb{R};\mathbb{G};\mathbb{P}} (\Lambda' \lhd \{\mathsf{t} \leadsto \mathbb{C}_2\}, \circ, \Gamma)\ .$$
Besides, since (C.5), $\mathsf{outf}(\Lambda, \mathsf{t})$, $\mathsf{outf}(\Lambda', \mathsf{t})$ and $\mathsf{fstep}(\mathbb{G})$, we know
$$((\sigma_o, \Lambda \lhd \{\mathsf{t} \leadsto \mathbb{C}_2\}), (\sigma_o', \Lambda' \lhd \{\mathsf{t} \leadsto \mathbb{C}_2\})) \in \mathbb{G} \uplus \mathsf{True}.$$
Finally, from (C.3), $\mathsf{outf}(\Lambda, \mathsf{t})$ and $\mathsf{outf}(\Lambda', \mathsf{t})$, we can prove:
$$(\Lambda \lhd \{\mathsf{t} \leadsto \mathbb{C}_2\}, \sigma_c, \circ) \overset{H}{\rightrightarrows}_{\mathsf{t},\Gamma} (\Lambda' \lhd \{\mathsf{t} \leadsto \mathbb{C}_2\}, \sigma_c', \circ)\ .$$

3. If $(C_1; C_2, (\sigma_c, \sigma_o, \circ)) \xrightarrow{e}_{\mathsf{t},\Pi} (C; \textbf{noret}, (\sigma_c', \sigma_o', \kappa'))$ and $\kappa' = (\sigma_l, x, (C_1'; C_2))$,
thus $(C_1, (\sigma_c, \sigma_o, \circ)) \xrightarrow{e}_{\mathsf{t},\Pi} (C; \textbf{noret}, (\sigma_c', \sigma_o', \kappa_1'))$ and $\kappa_1' = (\sigma_l, x, C_1')$.
From the first premise, we know there exist Λ', ak_1' and H such that
$$(\Lambda, \sigma_c, \circ) \overset{H}{\rightrightarrows}_{\mathsf{t},\Gamma} (\Lambda', \sigma_c', ak_1') \tag{C.7}$$
$$\mathsf{get\_obsv}(e) = H \tag{C.8}$$
$$((\sigma_o, \Lambda), (\sigma_o', \Lambda')) \in \mathbb{G} \uplus \mathsf{True} \tag{C.9}$$
$$(C; \textbf{noret}, (\sigma_c', \sigma_o', \kappa_1'), \Pi) \preceq^{\mathsf{t}}_{\mathbb{R};\mathbb{G};\mathbb{P}} (\Lambda', ak_1', \Gamma) \tag{C.10}$$
Since (C.9), $\mathsf{outf}(\Lambda, \mathsf{t})$ and $\mathsf{fstep}(\mathbb{G})$, we know
$$((\sigma_o, \Lambda \lhd \{\mathsf{t} \leadsto \mathbb{C}_2\}), (\sigma_o', \Lambda')) \in \mathbb{G} \uplus \mathsf{True}.$$
From (C.10), we know there exists $\mathbb{C}_1'$ such that $ak_1' = (x, \mathbb{C}_1')$. Let
$$ak' = (x, (\mathbb{C}_1'; \mathbb{C}_2)).$$
Then, from (C.7) and $\mathsf{outf}(\Lambda, \mathsf{t})$, we can prove:
$$(\Lambda \lhd \{\mathsf{t} \leadsto \mathbb{C}_2\}, \sigma_c, \circ) \overset{H}{\rightrightarrows}_{\mathsf{t},\Gamma} (\Lambda', \sigma_c', ak')\ .$$
Finally, by the following Lemma 30, we know
$$(C; \textbf{noret}, (\sigma_c', \sigma_o', \kappa'), \Pi) \preceq^{\mathsf{t}}_{\mathbb{R};\mathbb{G};\mathbb{P}} (\Lambda', ak', \Gamma)\ .$$

4. If $(C_1; C_2, (\sigma_c, \sigma_o, \circ)) \xrightarrow{e}_{\mathsf{t},\Pi} \textbf{abort}$,
by the operational semantics, we know
$$(C_1, (\sigma_c, \sigma_o, \circ)) \xrightarrow{e}_{\mathsf{t},\Pi} \textbf{abort}.$$
By the first premise, we know there exists H such that
$$(\Lambda, \sigma_c, \circ) \overset{H}{\rightrightarrows}_{\mathsf{t},\Gamma} \textbf{abort} \tag{C.11}$$
$$\mathsf{get\_obsv}(e) = H \tag{C.12}$$
From (C.11) and $\mathsf{outf}(\Lambda, \mathsf{t})$, we can prove:
$$(\Lambda \lhd \{\mathsf{t} \leadsto \mathbb{C}_2\}, \sigma_c, \circ) \overset{H}{\rightrightarrows}_{\mathsf{t},\Gamma} \textbf{abort}\ .$$

5. For any σ_c', σ_o' and Λ', if
$$((\sigma_o, \Lambda \lhd \{\mathsf{t} \leadsto \mathbb{C}_2\}), (\sigma_o', \Lambda')) \in \mathbb{R} \uplus \mathsf{Id},$$
since $\mathsf{fstep}(\mathbb{R})$ and $\mathsf{outf}(\Lambda, \mathsf{t})$, we know there exists Λ'' such that
$$\Lambda' = \Lambda'' \lhd \{\mathsf{t} \leadsto \mathbb{C}_2\}\ ,\ ((\sigma_o, \Lambda), (\sigma_o', \Lambda'')) \in \mathbb{R} \uplus \mathsf{Id}\ .$$
By the first premise, we know
$$(C_1, (\sigma_c', \sigma_o', \circ), \Pi) \preceq^{\mathsf{t}}_{\mathbb{R};\mathbb{G};\mathbb{P}} (\Lambda'', \circ, \Gamma)\ .$$
Also we have $\mathsf{outf}(\Lambda'', \mathsf{t})$. By the hypothesis, we know
$$(C_1; C_2, (\sigma_c', \sigma_o', \circ), \Pi) \preceq^{\mathsf{t}}_{\mathbb{R};\mathbb{G};\mathbb{P}} (\Lambda'' \lhd \{\mathsf{t} \leadsto \mathbb{C}_2\}, \circ, \Gamma).$$

By definition, we complete the proof. □

Lemma 30. If

1. $(C, (\sigma_c, \sigma_o, \kappa), \Pi) \preceq^{\mathsf{t}}_{\mathbb{R};\mathbb{G};\mathbb{P}} (\Lambda, ak, \Gamma)$,
2. $(C_2, \Pi) \preceq^{\mathsf{t}}_{\mathbb{R};\mathbb{G};\mathbb{P}} (\mathbb{C}_2, \Gamma)$,
3. $\kappa = (\sigma_l, x, C_1)$, $ak = (x, \mathbb{C}_1)$,
$\kappa' = (\sigma_l, x, (C_1; C_2))$, $ak' = (x, (\mathbb{C}_1; \mathbb{C}_2))$,

then $(C, (\sigma_c, \sigma_o, \kappa'), \Pi) \preceq^{\mathsf{t}}_{\mathbb{R};\mathbb{G};\mathbb{P}} (\Lambda, ak', \Gamma)$.

Proof: By definition and co-induction. We have the following cases:

1. If $(C, (\sigma_c, \sigma_o, \kappa')) \xrightarrow{e}_{\mathsf{t},\Pi} (C_1; C_2, (\sigma_c', \sigma_o, \circ))$,
thus $(C, (\sigma_c, \sigma_o, \kappa)) \xrightarrow{e}_{\mathsf{t},\Pi} (C_1, (\sigma_c', \sigma_o, \circ))$.

From the first premise, we know there exist Λ', ak'' and H such that

$$(\Lambda, \sigma_c, ak) \overset{H}{\rightrightarrows}_{\mathsf{t},\Gamma} (\Lambda', \sigma'_c, ak'') \tag{C.13}$$

$$\mathsf{get\_obsv}(e) = H \tag{C.14}$$

$$((\sigma_o, \Lambda), (\sigma_o, \Lambda')) \in \mathbb{G} \uplus \mathsf{True} \tag{C.15}$$

$$(C_1, (\sigma'_c, \sigma_o, \circ), \Pi) \preceq^{\mathsf{t}}_{\mathbb{R};\mathbb{G};\mathbb{P}} (\Lambda', ak'', \Gamma) \tag{C.16}$$

By (C.16), we know $ak'' = \circ$. Then from (C.13), we know $\mathsf{outf}(\Lambda', \mathsf{t})$. We can prove:

$$(\Lambda, \sigma_c, ak') \overset{H}{\rightrightarrows}_{\mathsf{t},\Gamma} (\Lambda' \triangleleft \{\mathsf{t} \rightsquigarrow \mathbb{C}_2\}, \sigma'_c, \circ).$$

Besides, since (C.15) and $\mathsf{fstep}(\mathbb{G})$, we know

$$((\sigma_o, \Lambda), (\sigma_o, \Lambda' \triangleleft \{\mathsf{t} \rightsquigarrow \mathbb{C}_2\})) \in \mathbb{G} \uplus \mathsf{True}.$$

Finally, by Lemma 29, we know

$$(C_1; C_2, (\sigma'_c, \sigma_o, \circ), \Pi) \preceq^{\mathsf{t}}_{\mathbb{R};\mathbb{G};\mathbb{P}} (\Lambda' \triangleleft \{\mathsf{t} \rightsquigarrow \mathbb{C}_2\}, \circ, \Gamma) \ .$$

2. If $(C, (\sigma_c, \sigma_o, \kappa')) \xrightarrow{e}_{\mathsf{t},\Pi} (C', (\sigma'_c, \sigma'_o, \kappa''))$ thus we know there exists σ'_l such that $\kappa'' = (\sigma'_l, x, (C_1; C_2))$, thus $(C, (\sigma_c, \sigma_o, \kappa)) \xrightarrow{e}_{\mathsf{t},\Pi} (C', (\sigma'_c, \sigma'_o, \kappa'''))$ and $\kappa''' = (\sigma'_l, x, C_1)$.
   From the first premise, we know there exist Λ', ak''' and H such that

$$(\Lambda, \sigma_c, ak) \overset{H}{\rightrightarrows}_{\mathsf{t},\Gamma} (\Lambda', \sigma'_c, ak''') \tag{C.17}$$

$$\mathsf{get\_obsv}(e) = H \tag{C.18}$$

$$((\sigma_o, \Lambda), (\sigma'_o, \Lambda')) \in \mathbb{G} \uplus \mathsf{True} \tag{C.19}$$

$$(C', (\sigma'_c, \sigma'_o, \kappa'''), \Pi) \preceq^{\mathsf{t}}_{\mathbb{R};\mathbb{G};\mathbb{P}} (\Lambda', ak''', \Gamma) \tag{C.20}$$

   By (C.20), we know there exists $\mathbb{C}'_1$ such that $ak''' = (x, \mathbb{C}'_1)$. Let $ak'' = (x, (\mathbb{C}'_1; \mathbb{C}_2))$. By the hypothesis, we know

$$(C', (\sigma'_c, \sigma'_o, \kappa''), \Pi) \preceq^{\mathsf{t}}_{\mathbb{R};\mathbb{G};\mathbb{P}} (\Lambda', ak'', \Gamma) \ .$$

   Besides, from (C.17), we can prove

$$(\Lambda, \sigma_c, ak') \overset{H}{\rightrightarrows}_{\mathsf{t},\Gamma} (\Lambda', \sigma'_c, ak'') \ .$$

3. If $(C, (\sigma_c, \sigma_o, \kappa')) \xrightarrow{e}_{\mathsf{t},\Pi} \textbf{abort}$, by the operational semantics, we know

$$(C, (\sigma_c, \sigma_o, \kappa)) \xrightarrow{e}_{\mathsf{t},\Pi} \textbf{abort}.$$

   By the first premise, we know there exists H such that

$$(\Lambda, \sigma_c, ak) \overset{H}{\rightrightarrows}_{\mathsf{t},\Gamma} \textbf{abort} \tag{C.21}$$

$$\mathsf{get\_obsv}(e) = H \tag{C.22}$$

   We can prove:

$$(\Lambda, \sigma_c, ak') \overset{H}{\rightrightarrows}_{\mathsf{t},\Gamma} \textbf{abort} \ .$$

4. For any σ'_c, σ'_o and Λ', if

$$((\sigma_o, \Lambda), (\sigma'_o, \Lambda')) \in \mathbb{R} \uplus \mathsf{Id},$$

   By the first premise, we know

$$(C, (\sigma'_c, \sigma'_o, \kappa), \Pi) \preceq^{\mathsf{t}}_{\mathbb{R};\mathbb{G};\mathbb{P}} (\Lambda', ak, \Gamma) \ .$$

   By the hypothesis, we know

$$(C, (\sigma'_c, \sigma'_o, \kappa'), \Pi) \preceq^{\mathsf{t}}_{\mathbb{R};\mathbb{G};\mathbb{P}} (\Lambda', ak', \Gamma).$$

By definition, we complete the proof. □

Lemma 31 (If-then-else Compositionality of Simulation). If $(C_1, \Pi) \preceq^{\mathsf{t}}_{\mathbb{R};\mathbb{G};\mathbb{P}} (\mathbb{C}_1, \Gamma)$ and $(C_2, \Pi) \preceq^{\mathsf{t}}_{\mathbb{R};\mathbb{G};\mathbb{P}} (\mathbb{C}_2, \Gamma)$, then $(\textbf{if}\ (B)\ C_1\ \textbf{else}\ C_2, \Pi) \preceq^{\mathsf{t}}_{\mathbb{R};\mathbb{G};\mathbb{P}} (\textbf{if}\ (B)\ \mathbb{C}_1\ \textbf{else}\ \mathbb{C}_2, \Gamma)$.

Proof: We want to prove: for any σ_o and Λ, if $(\sigma_o, \Lambda) \in \mathbb{P}$, then for any σ_c,

$$\begin{array}{l}(\textbf{if}\ (B)\ C_1\ \textbf{else}\ C_2, (\sigma_c, \sigma_o, \circ), \Pi) \preceq^{\mathsf{t}}_{\mathbb{R};\mathbb{G};\mathbb{P}} \\ (\Lambda * \{(\{\mathsf{t} \rightsquigarrow (\textbf{if}\ (B)\ \mathbb{C}_1\ \textbf{else}\ \mathbb{C}_2)\}, \emptyset)\}, \circ, \Gamma)\end{array}$$

We prove it by definition and co-induction. We have the following cases:

1. If $(\textbf{if}\ (B)\ C_1\ \textbf{else}\ C_2, (\sigma_c, \sigma_o, \circ)) \longrightarrow_{\mathsf{t},\Pi} (C_1, (\sigma_c, \sigma_o, \circ))$ where $[\![B]\!]_{\sigma_c} = \textbf{true}$,
   by the first premise, we know

$$(C_1, (\sigma_c, \sigma_o, \circ), \Pi) \preceq^{\mathsf{t}}_{\mathbb{R};\mathbb{G};\mathbb{P}} (\Lambda * \{(\{\mathsf{t} \rightsquigarrow \mathbb{C}_1\}, \emptyset)\}, \circ, \Gamma) \ .$$

   Also we have

$$\begin{array}{l}(\Lambda * \{(\{\mathsf{t} \rightsquigarrow (\textbf{if}\ (B)\ \mathbb{C}_1\ \textbf{else}\ \mathbb{C}_2)\}, \emptyset)\}, \sigma_c, \circ) \rightrightarrows_{\mathsf{t},\Gamma} \\ (\Lambda * \{(\{\mathsf{t} \rightsquigarrow \mathbb{C}_1\}, \emptyset)\}, \sigma_c, \circ) \ .\end{array}$$

   Also, since $\mathbb{P} \Rightarrow \mathbb{I}$, we know $(\sigma_o, \Lambda) \in \mathbb{I}$. Since $\mathbb{I} \triangleright \mathbb{G}$, we have:

$$((\sigma_o, \Lambda), (\sigma_o, \Lambda)) \in \mathbb{G} \ .$$

   Thus we have

$$\begin{array}{l}((\sigma_o, \Lambda * \{(\{\mathsf{t} \rightsquigarrow (\textbf{if}\ (B)\ \mathbb{C}_1\ \textbf{else}\ \mathbb{C}_2)\}, \emptyset)\}), \\ \quad (\sigma_o, \Lambda * \{(\{\mathsf{t} \rightsquigarrow \mathbb{C}_1\}, \emptyset)\})) \in \mathbb{G} \uplus \mathsf{True} \ .\end{array}$$

2. The case is similar when

$$(\textbf{if}\ (B)\ C_1\ \textbf{else}\ C_2, (\sigma_c, \sigma_o, \circ)) \longrightarrow_{\mathsf{t},\Pi} (C_2, (\sigma_c, \sigma_o, \circ))$$

   where $[\![B]\!]_{\sigma_c} = \textbf{false}$.

3. If $(\textbf{if}\ (B)\ C_1\ \textbf{else}\ C_2, (\sigma_c, \sigma_o, \circ)) \xrightarrow{e}_{\mathsf{t},\Pi} \textbf{abort}$, by the operational semantics, we know $[\![B]\!]_{\sigma_c}$ is undefined. Thus we have

$$(\Lambda * \{(\{\mathsf{t} \rightsquigarrow \textbf{if}\ (B)\ \mathbb{C}_1\ \textbf{else}\ \mathbb{C}_2\}, \emptyset)\}, \sigma_c, \circ) \overset{H}{\rightrightarrows}_{\mathsf{t},\Gamma} \textbf{abort}$$

   where $H = \mathsf{get\_obsv}(e)$.

4. For any σ'_c, σ'_o and Λ', if

$$\begin{array}{l}((\sigma_o, \Lambda * \{(\{\mathsf{t} \rightsquigarrow \textbf{if}\ (B)\ \mathbb{C}_1\ \textbf{else}\ \mathbb{C}_2\}, \emptyset)\}), (\sigma'_o, \Lambda')) \\ \in \mathbb{R} \uplus \mathsf{Id}\end{array}$$

   since $\mathsf{fstep}(\mathbb{R})$, $(\sigma_o, \Lambda) \in \mathbb{I}$ and $\mathbb{I} \triangleright \mathbb{R}$, we know there exists Λ'' such that

$$\begin{array}{l}\Lambda' = \Lambda'' * \{(\{\mathsf{t} \rightsquigarrow \textbf{if}\ (B)\ \mathbb{C}_1\ \textbf{else}\ \mathbb{C}_2\}, \emptyset)\}) \ , \\ ((\sigma_o, \Lambda), (\sigma'_o, \Lambda'')) \in \mathbb{R} \ .\end{array}$$

   Since $\mathsf{Sta}(\mathbb{P}, \mathbb{R})$, we know

$$(\sigma'_o, \Lambda'') \in \mathbb{P} \ .$$

   From the hypothesis, we get

$$\begin{array}{l}(\textbf{if}\ (B)\ C_1\ \textbf{else}\ C_2, (\sigma'_c, \sigma'_o, \circ), \Pi) \preceq^{\mathsf{t}}_{\mathbb{R};\mathbb{G};\mathbb{P}} \\ (\Lambda'' * \{(\{\mathsf{t} \rightsquigarrow (\textbf{if}\ (B)\ \mathbb{C}_1\ \textbf{else}\ \mathbb{C}_2)\}, \emptyset)\}, \circ, \Gamma) \ .\end{array}$$

   This case can also be proved by unfolding the premises and then applying the new hypothesis, without using $\mathsf{Sta}(\mathbb{P}, \mathbb{R})$.

By definition, we complete the proof. □

Lemma 32 (While-loop Compositionality of Simulation). If $(C, \Pi) \preceq^{\mathsf{t}}_{\mathbb{R};\mathbb{G};\mathbb{P}} (\mathbb{C}, \Gamma)$, then $(\textbf{while}\ (B)\{C\}, \Pi) \preceq^{\mathsf{t}}_{\mathbb{R};\mathbb{G};\mathbb{P}} (\textbf{while}\ (B)\{\mathbb{C}\}, \Gamma)$.

Proof: The proof is similar to the proof of Lemma 31, by applying Lemmas 26 and 29. □

Lemma 33 (Parallel Compositionality of Simulation). If for any $i \in \{1, \ldots, n\}$, we have $(C_i, \Pi) \preceq^{i}_{\mathbb{R}_i;\mathbb{G}_i;\mathbb{P}_i} (\mathbb{C}_i, \Gamma)$, $\mathbb{R}_i = \bigvee_{j \neq i} \mathbb{G}_j$, $\mathbb{I} \triangleright \{\mathbb{R}_i, \mathbb{G}_i\}$, $\mathbb{P}_i \Rightarrow \mathbb{I}$, $\lfloor \widetilde{p} \rfloor \Rightarrow \mathbb{P}_i$, then $(\textbf{let}\ \Pi\ \textbf{in}\ C_1 \parallel \ldots \parallel C_n) \preceq_{\widetilde{p}} (\textbf{with}\ \Gamma\ \textbf{do}\ \mathbb{C}_1 \parallel \ldots \parallel \mathbb{C}_n)$.

Proof: We want to prove: for any σ_c, σ_o, θ, $\mathcal{K}$ and $\mathbb{K}$, if $(\sigma_o, \theta) \in \widetilde{p}$, $\forall \mathsf{t}.\ \mathcal{K}(\mathsf{t}) = \circ$ and $\forall \mathsf{t}.\ \mathbb{K}(\mathsf{t}) = \circ$, then

$$\begin{array}{l}(\textbf{let}\ \Pi\ \textbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n, (\sigma_c, \sigma_o, \mathcal{K})) \preceq \\ \{(\textbf{with}\ \Gamma\ \textbf{do}\ \mathbb{C}_1 \,\|\, \ldots \,\|\, \mathbb{C}_n, (\sigma_c, \theta, \mathbb{K}))\}\,.\end{array}$$

For any i, since $\lfloor \widetilde{p} \rfloor \Rightarrow \mathbb{P}_i$, we know

$$(\sigma_o, \{(\emptyset, \theta)\}) \in \mathbb{P}_i\,.$$

From the premise, we know

$$(C_i, (\sigma_c, \sigma_o, \circ), \Pi) \preceq^i_{\mathbb{R}_i;\mathbb{G}_i;\mathbb{P}_i} (\{(\emptyset, \theta)\} * \{(\{i \rightsquigarrow \mathbb{C}_i\}, \emptyset)\}, \circ, \Gamma)\,,$$

which is reduced to:

$$(C_i, (\sigma_c, \sigma_o, \circ), \Pi) \preceq^i_{\mathbb{R}_i;\mathbb{G}_i;\mathbb{P}_i} (\{(\{i \rightsquigarrow \mathbb{C}_i\}, \theta)\}, \circ, \Gamma)\,.$$

On the other hand, since $\mathbb{P}_i \Rightarrow \mathbb{I}$, we know

$$(\sigma_o, \{(\emptyset, \theta)\}) \in \mathbb{I}\,.$$

From the following Lemma 34, we know

$$\begin{array}{l}(\textbf{let}\ \Pi\ \textbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n, (\sigma_c, \sigma_o, \mathcal{K})) \preceq \\ (\{(\emptyset, \theta)\} * (\bigotimes_{i \in [1..n]} \{(\{i \rightsquigarrow \mathbb{C}_i\}, \emptyset)\}), \sigma_c, \mathbb{K})_\Gamma\,.\end{array}$$

Here we define

$$\begin{array}{l}(\Lambda, \sigma_c, \mathbb{K})_\Gamma \ \stackrel{\text{def}}{=}\ \{(\textbf{with}\ \Gamma\ \textbf{do}\ [\![\mathbb{U}]\!], (\sigma_c, \theta, \mathbb{K})) \mid (\mathbb{U}, \theta) \in \Lambda\} \\ \quad \text{where } [\![\mathbb{U}]\!] \ \stackrel{\text{def}}{=}\ \mathbb{U}(1) \,\|\, \ldots \,\|\, \mathbb{U}(n)\end{array} \tag{C.23}$$

Thus

$$\begin{array}{l}(\{(\emptyset, \theta)\} * (\bigotimes_{i \in [1..n]} \{(\{i \rightsquigarrow \mathbb{C}_i\}, \emptyset)\}), \sigma_c, \mathbb{K})_\Gamma \\ = (\{(\{i \rightsquigarrow \mathbb{C}_i \mid i \in [1..n]\}, \theta)\}, \sigma_c, \mathbb{K})_\Gamma \\ = \{(\textbf{with}\ \Gamma\ \textbf{do}\ \mathbb{C}_1 \,\|\, \ldots \,\|\, \mathbb{C}_n, (\sigma_c, \theta, \mathbb{K}))\}\,.\end{array}$$

Thus we get the conclusion. □

Lemma 34. If for any $i \in \{1, \ldots, n\}$, we have

1. $(C_i, (\sigma_c, \sigma_o \uplus \sigma_i, \kappa_i), \Pi) \preceq^i_{\mathbb{R}_i;\mathbb{G}_i;\mathbb{P}_i} (\Lambda * \Lambda_i, ak_i, \Gamma)$,
2. $(\sigma_o, \Lambda) \in \mathbb{I}$, $\ dom(\Lambda * (\bigotimes_i \Lambda_i).\mathbb{U}) = [1..n]$,
3. $\mathbb{R}_i = \bigvee_{j \neq i} \mathbb{G}_j$, $\mathbb{I} \triangleright \{\mathbb{R}_i, \mathbb{G}_i\}$,
4. $\mathcal{K} = \{i \rightsquigarrow \kappa_i \mid i \in [1..n]\}$, $\ \mathbb{K} = \{i \rightsquigarrow ak_i \mid i \in [1..n]\}$,

then

$$\begin{array}{l}(\textbf{let}\ \Pi\ \textbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n, (\sigma_c, \sigma_o \uplus (\biguplus_i \sigma_i), \mathcal{K})) \preceq \\ (\Lambda * (\bigotimes_i \Lambda_i), \sigma_c, \mathbb{K})_\Gamma\,,\end{array}$$

where $(\Lambda, \sigma_c, \mathbb{K})_\Gamma$ is defined in (C.23).

Proof: By definition and co-induction. Let

$$\sigma = \sigma_o \uplus (\biguplus_i \sigma_i) \ \text{ and } \ \Omega = (\Lambda * (\bigotimes_i \Lambda_i), \sigma_c, \mathbb{K})_\Gamma.$$

We have two cases:

1. If $(\textbf{let}\ \Pi\ \textbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n, (\sigma_c, \sigma, \mathcal{K})) \xrightarrow{e} (W', \mathcal{S}')$, we have the following cases:
   (a) If $C_1 = \ldots = C_n = \textbf{skip}$,
   then $e = \tau$ (we use τ to denote an empty (silent) event), $W' = \textbf{skip}$ and $\mathcal{S}' = (\sigma_c, \sigma, \mathcal{K})$.
   From the premise, we know
   $$\forall i.\ \kappa_i = \circ,\ ak_i = \circ$$
   and there exists Λ' such that
   $$\Lambda * \Lambda_i = \Lambda' * \{(\{i \rightsquigarrow \textbf{skip}\}, \emptyset)\}\,.$$
   Thus for any i, $(\Lambda * (\bigotimes_i \Lambda_i).\mathbb{U})(i) = \textbf{skip}$. Thus for any $\mathbb{W}$, if $(\mathbb{W}, \_) \in \Omega$, then $\mathbb{W} = \textbf{with}\ \Gamma\ \textbf{do}\ \textbf{skip} \,\|\, \ldots \,\|\, \textbf{skip}$.
   Let $\Omega' = \{(\textbf{skip}, \mathbb{S}) \mid (\_, \mathbb{S}) \in \Omega\}$. Thus $\Omega \rightrightarrows \Omega'$ and $(W', \mathcal{S}') \preceq \Omega'$.
   (b) If $(C_i, (\sigma_c, \sigma, \kappa_i)) \xrightarrow{e}_{i,\Pi} (C_i', (\sigma_c', \sigma'', \kappa_i'))$, by locality of concrete code, we know one of the following holds:
      i. If $(C_i, (\sigma_c, \sigma_o \uplus \sigma_i, \kappa_i)) \xrightarrow{e'}_{i,\Pi} \textbf{abort}$,
      from the premise we know: $\kappa_i = \circ$.
      Then by the concrete operational semantics, we know $(C_i, (\sigma_c, \sigma, \circ)) \xrightarrow{e'}_{i,\Pi} \textbf{abort}$, which leads to a contradiction.
      ii. If $(C_i, (\sigma_c, \sigma_o \uplus \sigma_i, \kappa_i)) \xrightarrow{e}_{i,\Pi} (C_i', (\sigma_c', \sigma', \kappa_i'))$,
      thus $\sigma'' = \sigma' \uplus (\biguplus_{j \neq i} \sigma_j)$.
      From the premise, we know there exist Λ', ak_i' and H such that
      $$(\Lambda * \Lambda_i, \sigma_c, ak_i) \overset{H}{\rightrightarrows}_{i,\Gamma} (\Lambda', \sigma_c', ak_i') \tag{C.24}$$
      $$\mathsf{get\_obsv}(e) = H \tag{C.25}$$
      $$((\sigma_o \uplus \sigma_i, \Lambda * \Lambda_i), (\sigma', \Lambda')) \in \mathbb{G}_i \uplus \mathsf{True} \tag{C.26}$$
      $$(C_i', (\sigma_c', \sigma', \kappa_i'), \Pi) \preceq^i_{\mathbb{R}_i;\mathbb{G}_i;\mathbb{P}_i} (\Lambda', ak_i', \Gamma) \tag{C.27}$$
      First, from (C.26) and $\mathbb{I} \triangleright \{\mathbb{R}_i, \mathbb{G}_i\}$, we know
      $$((\sigma_o \uplus \sigma_i, \Lambda * \Lambda_i), (\sigma', \Lambda')) \in (\mathbb{I} \ltimes \mathbb{I}) \uplus \mathsf{True}$$
      Since $(\sigma_o, \Lambda) \in \mathbb{I}$, we know there exist unique σ_o' and Λ'' such that
      $$((\sigma_o, \Lambda), (\sigma_o', \Lambda'')) \in \mathbb{G}_i\,, \qquad (\sigma_o', \Lambda'') \in \mathbb{I}$$
      and there exist σ_i' and Λ_i' such that
      $$\sigma' = \sigma_o' \uplus \sigma_i', \qquad \Lambda' = \Lambda'' * \Lambda_i'.$$
      Thus (C.27) can be rewritten as:
      $$(C_i', (\sigma_c', \sigma_o' \uplus \sigma_i', \kappa_i'), \Pi) \preceq^i_{\mathbb{R}_i;\mathbb{G}_i;\mathbb{P}_i} (\Lambda'' * \Lambda_i', ak_i', \Gamma)$$
      On the other hand, for all j such that $j \neq i$,
      since $\mathbb{R}_j = \bigvee_{k \neq j} \mathbb{G}_k$, we know $\mathbb{G}_i \Rightarrow \mathbb{R}_j$. Thus
      $$((\sigma_o \uplus \sigma_j, \Lambda * \Lambda_j), (\sigma_o' \uplus \sigma_j, \Lambda'' * \Lambda_j)) \in \mathbb{R}_j \uplus \mathsf{Id}$$
      From the premise, we know
      $$(C_j, (\sigma_c', \sigma_o' \uplus \sigma_j, \kappa_j), \Pi) \preceq^j_{\mathbb{R}_j;\mathbb{G}_j;\mathbb{P}_j} (\Lambda'' * \Lambda_j, ak_j, \Gamma)$$
      Besides, from (C.24), we know
      $$dom((\Lambda * \Lambda_i).\mathbb{U}) = dom((\Lambda'' * \Lambda_i').\mathbb{U})$$
      Let $\mathcal{K}' = \mathcal{K}\{i \rightsquigarrow \kappa_i'\}$ and $\mathbb{K}' = \mathbb{K}\{i \rightsquigarrow ak_i'\}$. Then by the hypothesis, we know
      $$\begin{array}{l}(\textbf{let}\ \Pi\ \textbf{in}\ C_1 \,\|\, \ldots C_i' \ldots \,\|\, C_n, (\sigma_c', \sigma'', \mathcal{K}')) \preceq \\ (\Lambda'' * \Lambda_i' * (\bigotimes_{j \neq i} \Lambda_j), \sigma_c', \mathbb{K}')_\Gamma\end{array}$$
      Let $\Omega' = (\Lambda'' * \Lambda_i' * (\bigotimes_{j \neq i} \Lambda_j), \sigma_c', \mathbb{K}')_\Gamma$.
      Secondly, we prove $\Omega \overset{H}{\rightrightarrows} \Omega'$ by its definition.
      For any $\mathbb{W}'$ and $\mathbb{S}'$ such that $(\mathbb{W}', \mathbb{S}') \in \Omega'$, we know there exist θ' and $\mathbb{U}'$ such that
      $$\begin{array}{l}\mathbb{W}' = \textbf{with}\ \Gamma\ \textbf{do}\ [\![\mathbb{U}']\!]\,, \\ \mathbb{S}' = (\sigma_c', \theta', \mathbb{K}')\,, \\ (\mathbb{U}', \theta') \in \Lambda'' * \Lambda_i' * (\bigotimes_{j \neq i} \Lambda_j)\,.\end{array}$$
      From (C.24), we know
      $$\begin{array}{l}\forall \mathbb{U}'', \theta''.\ (\mathbb{U}'', \theta'') \in \Lambda'' * \Lambda_i' \\ \implies \exists \mathbb{U}, \theta, H'.\ (\mathbb{U}, \theta) \in \Lambda * \Lambda_i \\ \qquad \wedge\ (\mathbb{U}, (\sigma_c, \theta, ak_i)) \overset{H'}{\dashrightarrow}{}^*_{i,\Gamma} (\mathbb{U}'', (\sigma_c', \theta'', ak_i')) \\ \qquad \wedge\ \mathsf{get\_obsv}(H') = H\end{array}$$
      By locality of abstract operations, we know: there exist $\mathbb{U}$, θ and H' such that
      $$(\mathbb{U}, \theta) \in \Lambda * \Lambda_i * (\bigotimes_{j \neq i} \Lambda_j) \tag{C.28}$$
      $$(\mathbb{U}, (\sigma_c, \theta, ak_i)) \overset{H'}{\dashrightarrow}{}^*_{i,\Gamma} (\mathbb{U}', (\sigma_c', \theta', ak_i')) \tag{C.29}$$
      $$\mathsf{get\_obsv}(H') = H \tag{C.30}$$
      Let
      $$\mathbb{W} = \textbf{with}\ \Gamma\ \textbf{do}\ [\![\mathbb{U}]\!]\,, \qquad \mathbb{S} = (\sigma_c, \theta, \mathbb{K})\,.$$

From (C.28) and definition (C.23), we know $(\mathbb{W}, \mathbb{S}) \in \Omega$.
From (C.29), we can prove:

$$(\textbf{with } \Gamma \textbf{ do } [\![\mathbb{U}]\!], (\sigma_c, \theta, \mathbb{K})) \overset{H'}{\Longrightarrow}{}^* (\textbf{with } \Gamma \textbf{ do } [\![\mathbb{U}']\!], (\sigma_c', \theta', \mathbb{K}\{i \rightsquigarrow ak_i'\}))$$

which is $(\mathbb{W}, \mathbb{S}) \overset{H'}{\Longrightarrow}{}^* (\mathbb{W}', \mathbb{S}')$. Thus $\Omega \overset{H}{\rightrightarrows} \Omega'$.
So we finish this case.

2. If $(\textbf{let } \Pi \textbf{ in } C_1 \parallel \ldots \parallel C_n, (\sigma_c, \sigma, \mathcal{K})) \overset{e}{\longmapsto} \textbf{abort}$, by the operational semantics, we know there exists i such that

$$(C_i, (\sigma_c, \sigma, \kappa_i)) \overset{e}{\longrightarrow}_{i,\Pi} \textbf{abort}$$

By locality of the concrete code, we know

$$(C_i, (\sigma_c, \sigma_o \uplus \sigma_i, \kappa_i)) \overset{e}{\longrightarrow}_{i,\Pi} \textbf{abort}$$

From the premise, we know $\kappa_i = \circ$, $ak_i = \circ$ and there exists H such that $(\Lambda * \Lambda_i, \sigma_c, ak_i) \overset{H}{\rightrightarrows}_{i,\Gamma} \textbf{abort}$ and $\mathsf{get\_obsv}(e) = H$. Thus by locality of abstract operations we know: there exist Λ', ak_i' and H' such that

$$(\Lambda * \Lambda_i * (\bigotimes_{j \neq i} \Lambda_j), \sigma_c, ak_i) \overset{H'}{\rightrightarrows}_{i,\Gamma} (\Lambda' * (\bigotimes_{j \neq i} \Lambda_j), \sigma_c', ak_i') \tag{C.31}$$

and there exist $\mathbb{U}'$, θ' and H'' such that

$$(\mathbb{U}', \theta') \in \Lambda' * (\bigotimes_{j \neq i} \Lambda_j) \tag{C.32}$$

$$(\mathbb{U}'(i), (\sigma_c', \theta', ak_i')) \overset{H''}{\circ\!\!\longrightarrow}{}^*_{i,\Gamma} \textbf{abort} \tag{C.33}$$

$$\mathsf{get\_obsv}(H' :: H'') = H \tag{C.34}$$

From (C.31), we know there exist $\mathbb{U}$, θ and H''' such that

$$(\mathbb{U}, \theta) \in \Lambda * \Lambda_i * (\bigotimes_{j \neq i} \Lambda_j) \tag{C.35}$$

$$(\mathbb{U}, (\sigma_c, \theta, ak_i)) \overset{H'''}{\dashrightarrow}{}^*_{i,\Gamma} (\mathbb{U}', (\sigma_c', \theta', ak_i')) \tag{C.36}$$

$$\mathsf{get\_obsv}(H''') = H' \tag{C.37}$$

Let

$$\mathbb{W} = \textbf{with } \Gamma \textbf{ do } [\![\mathbb{U}]\!], \qquad \mathbb{S} = (\sigma_c, \theta, \mathbb{K}).$$

From (C.35) and definition (C.23), we know $(\mathbb{W}, \mathbb{S}) \in \Omega$.
From (C.33) and (C.36), we can prove:

$$(\textbf{with } \Gamma \textbf{ do } [\![\mathbb{U}]\!], (\sigma_c, \theta, \mathbb{K})) \overset{H''' :: H''}{\Longrightarrow}{}^* \textbf{abort}$$

which is $(\mathbb{W}, \mathbb{S}) \overset{H''' :: H''}{\Longrightarrow}{}^* \textbf{abort}$.
From (C.34) and (C.37), we know $\mathsf{get\_obsv}(H''' :: H'') = H$.
So we finish this case.

By definition, we complete the proof. □

C.1.3 Simulation for Program Implies Refinement

Lemma 35 (Simulation for Program Implies Refinement). For any W, $\mathbb{W}$ and $\widetilde{p}$, if $W \preceq_{\widetilde{p}} \mathbb{W}$, then

$$\forall \sigma_c, \sigma_o, \theta.\ (\sigma_o, \theta) \in \widetilde{p} \implies \mathcal{O}[\![W, (\sigma_c, \sigma_o)]\!] \subseteq \mathcal{O}[\![\mathbb{W}, (\sigma_c, \theta)]\!].$$

Proof: Immediate by applying the following Lemma 36. □

We overload the notation and define the following:

$$\mathcal{O}[\![W, \mathcal{S}]\!] \stackrel{\text{def}}{=} \{\mathsf{get\_obsv}(H) \mid \exists W', \mathcal{S}'.\ ((W, \mathcal{S}) \overset{H}{\longmapsto}{}^* (W', \mathcal{S}') \vee (W, \mathcal{S}) \overset{H}{\longmapsto}{}^* \textbf{abort})\}$$

$$\mathcal{O}[\![\mathbb{W}, \mathbb{S}]\!] \stackrel{\text{def}}{=} \{\mathsf{get\_obsv}(H) \mid \exists \mathbb{W}', \mathbb{S}'.\ ((\mathbb{W}, \mathbb{S}) \overset{H}{\Longrightarrow}{}^* (\mathbb{W}', \mathbb{S}') \vee (\mathbb{W}, \mathbb{S}) \overset{H}{\Longrightarrow}{}^* \textbf{abort})\}$$

Lemma 36. If $(W, \mathcal{S}) \preceq \Omega$, then there exist $\mathbb{W}$ and $\mathbb{S}$ such that $(\mathbb{W}, \mathbb{S}) \in \Omega$ and $\mathcal{O}[\![W, \mathcal{S}]\!] \subseteq \mathcal{O}[\![\mathbb{W}, \mathbb{S}]\!]$.

Proof: For any H such that $H \in \mathcal{O}[\![W, \mathcal{S}]\!]$, we know one of the following holds:

1. There exist W', $\mathcal{S}'$ and H' such that

$$(W, \mathcal{S}) \overset{H'}{\longmapsto}{}^* (W', \mathcal{S}'), \qquad H = \mathsf{get\_obsv}(H').$$

By the following Lemma 37, we know there exists Ω' such that

$$(W', \mathcal{S}') \preceq \Omega', \qquad \Omega \overset{H}{\rightrightarrows} \Omega'.$$

Thus

$$\forall \mathbb{W}', \mathbb{S}'.\ (\mathbb{W}', \mathbb{S}') \in \Omega' \implies \exists \mathbb{W}, \mathbb{S}, H'.\ (\mathbb{W}, \mathbb{S}) \in \Omega \wedge (\mathbb{W}, \mathbb{S}) \overset{H'}{\Longrightarrow}{}^* (\mathbb{W}', \mathbb{S}') \wedge \mathsf{get\_obsv}(H') = H$$

Thus $\exists \mathbb{W}, \mathbb{S}.\ (\mathbb{W}, \mathbb{S}) \in \Omega \wedge H \in \mathcal{O}[\![\mathbb{W}, \mathbb{S}]\!]$.

2. There exist W', $\mathcal{S}'$ and H' such that

$$(W, \mathcal{S}) \overset{H'}{\longmapsto}{}^* (W', \mathcal{S}'), \qquad (W', \mathcal{S}') \overset{e}{\longmapsto} \textbf{abort},$$

and $H = \mathsf{get\_obsv}(H' :: e)$. Let $H_1 = \mathsf{get\_obsv}(H')$.
By the following Lemma 37, we know there exists Ω' such that

$$(W', \mathcal{S}') \preceq \Omega', \qquad \Omega \overset{H_1}{\rightrightarrows} \Omega'.$$

Also we know there exist $\mathbb{W}'$, $\mathbb{S}'$ and H'' such that

$$(\mathbb{W}', \mathbb{S}') \in \Omega', \qquad (\mathbb{W}', \mathbb{S}') \overset{H''}{\Longrightarrow}{}^* \textbf{abort}, \quad \mathsf{get\_obsv}(e) = \mathsf{get\_obsv}(H'')$$

And, there exist $\mathbb{W}$, $\mathbb{S}$ and H''' such that

$$(\mathbb{W}, \mathbb{S}) \in \Omega, \qquad (\mathbb{W}, \mathbb{S}) \overset{H'''}{\Longrightarrow}{}^* (\mathbb{W}', \mathbb{S}'), \quad \mathsf{get\_obsv}(H''') = H_1$$

Thus we have:

$$(\mathbb{W}, \mathbb{S}) \overset{H''' :: H''}{\Longrightarrow}{}^* \textbf{abort}, \quad \mathsf{get\_obsv}(H''' :: H'') = H$$

Thus $H \in \mathcal{O}[\![\mathbb{W}, \mathbb{S}]\!]$.

Thus we get the conclusion. □

Lemma 37. For any n, if $(W, \mathcal{S}) \overset{H}{\longmapsto}{}^n (W', \mathcal{S}')$, $(W, \mathcal{S}) \preceq \Omega$, $H' = \mathsf{get\_obsv}(H)$, then there exists Ω' such that $(W', \mathcal{S}') \preceq \Omega'$ and $\Omega \overset{H'}{\rightrightarrows} \Omega'$.

Proof: By induction over n.
Base Case: $n = 0$. Thus $W' = W$, $\mathcal{S}' = \mathcal{S}$ and $H' = \epsilon$. We take $\Omega' = \Omega$.
Inductive Step: $n = k + 1$. Thus there exist W'', $\mathcal{S}''$, H_1 and e such that

$$(W, \mathcal{S}) \overset{e}{\longmapsto} (W'', \mathcal{S}''), \quad (W'', \mathcal{S}'') \overset{H_1}{\longmapsto}{}^k (W', \mathcal{S}'),$$

and $H = e :: H_1$. Let $H_1' = \mathsf{get\_obsv}(H_1)$.
By $(W, \mathcal{S}) \preceq \Omega$, we know there exist Ω'' and H'' such that $\Omega \overset{H''}{\rightrightarrows} \Omega''$, $\mathsf{get\_obsv}(e) = H''$ and $(W'', \mathcal{S}'') \preceq \Omega''$.
By the induction hypothesis, we know there exists Ω' such that

$(W', \mathcal{S}') \preceq \Omega'$ and $\Omega'' \overset{H_1'}{\rightrightarrows} \Omega'$.

Thus we know $\Omega \overset{H'}{\rightrightarrows} \Omega'$. □

C.2 Proofs of Lemma 9 (Logic Ensures Simulation for Method)

Below we first define the standard rely-guarantee-style judgment semantics. We derive the simulation for method (Definition 7) from the standard judgment semantics, and then prove that all the inference rules in Figure 11 are soundness *w.r.t.* this standard semantics.

C.2.1 Derive Simulation from Semantics of Judgments

Definition 38 (Semantics of Sequential Judgment). $\models_{\mathsf{t}} \{p\}\widetilde{C}\{q\}$ iff, for any Σ, if $\Sigma \models p$, the following are true:

1. for any Σ', if $(\widetilde{C}, \Sigma) \hookrightarrow_{\mathsf{t}}^* (\mathbf{skip}, \Sigma')$, then $\Sigma' \models q$;
2. $(\widetilde{C}, \Sigma) \not\hookrightarrow_{\mathsf{t}}^* \mathbf{abort}$.

Definition 39 (Semantics of Rely-Guarantee-Style Judgment). $R, G, I \models_{\mathsf{t}} \{p\}\widetilde{C}\{q\}$ iff, for any Σ, if $\Sigma \models p$, the following are true (where $\mathcal{R} = [\![R * \mathsf{Id}]\!]$ and $\mathcal{G} = [\![G * \mathsf{True}]\!]$):

1. for any Σ', if $(\widetilde{C}, \Sigma) \overset{\mathcal{R}}{\hookrightarrow}_{\mathsf{t}}^* (\mathbf{skip}, \Sigma')$, then $\Sigma' \models q$;
2. for any n, $(\widetilde{C}, \Sigma, \mathcal{R})$ $\mathsf{guar}_{\mathsf{t}}^n$ $\mathcal{G}$.

Here $[\![\, R \,]\!] \overset{\text{def}}{=} \{(\Sigma, \Sigma') \mid (\Sigma, \Sigma') \models R\}$.
The property $(\widetilde{C}, \Sigma, \mathcal{R})$ $\mathsf{guar}_{\mathsf{t}}^n$ $\mathcal{G}$ is inductively defined as follows:

1. $(\widetilde{C}, \Sigma, \mathcal{R})$ $\mathsf{guar}_{\mathsf{t}}^0$ $\mathcal{G}$ always holds;
2. $(\widetilde{C}, \Sigma, \mathcal{R})$ $\mathsf{guar}_{\mathsf{t}}^{k+1}$ $\mathcal{G}$ iff
   (a) $(\widetilde{C}, \Sigma) \not\hookrightarrow_{\mathsf{t}} \mathbf{abort}$;
   (b) for any Σ', if $(\Sigma, \Sigma') \in \mathcal{R}$, then $(\widetilde{C}, \Sigma', \mathcal{R})$ $\mathsf{guar}_{\mathsf{t}}^k$ $\mathcal{G}$;
   (c) for any $\widetilde{C}'$ and Σ', if $(\widetilde{C}, \Sigma) \hookrightarrow_{\mathsf{t}} (\widetilde{C}', \Sigma')$, then $(\Sigma, \Sigma') \in \mathcal{G}$ and $(\widetilde{C}', \Sigma', \mathcal{R})$ $\mathsf{guar}_{\mathsf{t}}^k$ $\mathcal{G}$.

Our logic is sound *w.r.t.* this standard semantics, as shown in the following theorem. We will prove this theorem in Appendix C.2.2.

Theorem 40 (Logic Soundness as Rely-Guarantee Reasoning). If $R, G, I \vdash_{\mathsf{t}} \{p\}\widetilde{C}\{q\}$, then $R, G, I \models_{\mathsf{t}} \{p\}\widetilde{C}\{q\}$.

Besides, as in LRG [8], we have the following property about I and the syntactic judgment.

Lemma 41. If $R, G, I \vdash_{\mathsf{t}} \{p\}\widetilde{C}\{q\}$, then $I \triangleright \{R, G\}$ and $p \vee q \Rightarrow I * \mathsf{true}$.

Proof: By induction over the derivation of the judgment $R, G, I \vdash_{\mathsf{t}} \{p\}\widetilde{C}\{q\}$. □

To prove Lemma 9, we first prove several lemmas which relate the instrumented semantics to the concrete semantics and the speculative steps. The erasure function is formally defined in Figure 13.

Lemma 42 (From Concrete to Instrumented Steps). For any C, σ, Δ, C', σ', t and $\widetilde{C}$, if

1. $(C, \sigma) \longrightarrow_{\mathsf{t}} (C', \sigma')$,
2. $\mathsf{Er}(\widetilde{C}) = C$, where $C \neq \mathbf{E}[\,\mathbf{return}\ \_\,]$,
3. $(\widetilde{C}, (\sigma, \Delta)) \not\hookrightarrow_{\mathsf{t}} \mathbf{abort}$,

then there exist $\widetilde{C}'$ and Δ' such that

1. $(\widetilde{C}, (\sigma, \Delta)) \hookrightarrow_{\mathsf{t}} (\widetilde{C}', (\sigma', \Delta'))$, and
2. $\mathsf{Er}(\widetilde{C}') = C'$.

Proof: By case analysis of $(C, \sigma) \longrightarrow_{\mathsf{t}} (C', \sigma')$.

1. $C = \mathbf{E}[\,\mathbf{skip}\,]$
   From premise 1, we know $C = (\mathbf{skip}; C')$ and $\sigma' = \sigma$.
   Since we assume instrumented commands are all inserted into atomic blocks, we know there exists $\widetilde{C}'$ such that $\widetilde{C} = (\mathbf{skip}; \widetilde{C}')$ and $\mathsf{Er}(\widetilde{C}') = C'$. Thus $(\widetilde{C}, (\sigma, \Delta)) \hookrightarrow_{\mathsf{t}} (\widetilde{C}', (\sigma, \Delta))$.
2. $C = \mathbf{E}[\,c\,]$
   From premise 1, we know $C' = \mathbf{E}[\,\mathbf{skip}\,]$.
   From premise 2, we know $\widetilde{C}' = \mathbf{E}'[\,c\,]$ where $\mathsf{Er}(\mathbf{E}') = \mathbf{E}$.
   Let $\widetilde{C}' = \mathbf{E}'[\,\mathbf{skip}\,]$ and $\Delta' = \Delta$, thus the conclusion holds.
3. $C = \mathbf{E}[\,\langle C_1 \rangle\,]$
   From premise 1 and the operational semantics, we know
   $$(C_1, \sigma) \longrightarrow_{\mathsf{t}}^* (\mathbf{skip}, \sigma')$$
   Since there are no nested atomic blocks, we get the conclusion by the following Lemma 43.
4. $C = \mathbf{E}[\,\mathbf{if}\ (B)\ C_1\ \mathbf{else}\ C_2\,]$
   From premise 1, we know $\sigma' = \sigma$ and
   $$\text{either } C' = \mathbf{E}[C_1\,] \text{ and } [\![B]\!]_\sigma = \mathbf{true}, \quad \text{or } C' = \mathbf{E}[C_2\,] \text{ and } [\![B]\!]_\sigma = \mathbf{false}.$$
   From premise 2, we know $\widetilde{C}' = \mathbf{E}'[\,\mathbf{if}\ (B)\ \widetilde{C}_1\ \mathbf{else}\ \widetilde{C}_2\,]$ where $\mathsf{Er}(\widetilde{C}_1) = C_1$, $\mathsf{Er}(\widetilde{C}_2) = C_2$ and $\mathsf{Er}(\mathbf{E}') = \mathbf{E}$. Thus we can prove the conclusion holds.
5. $C = \mathbf{E}[\,\mathbf{while}\ (B)\{C_1\}\,]$
   The case is similar to previous cases.

So we finish the proof. □

Lemma 43. For any n, if

1. $(C, \sigma) \longrightarrow_{\mathsf{t}}^n (\mathbf{skip}, \sigma')$,
2. $\mathsf{Er}(\widetilde{C}) = C$, where C does not have atomic blocks or returns,
3. $(\widetilde{C}, (\sigma, \Delta)) \not\hookrightarrow_{\mathsf{t}}^* \mathbf{abort}$,

then there exists Δ' such that $(\widetilde{C}, (\sigma, \Delta)) \hookrightarrow_{\mathsf{t}}^* (\mathbf{skip}, (\sigma', \Delta'))$.

Proof: By induction over n.
Base Case: $n = 0$, thus $C = \mathbf{skip}$ and $\sigma' = \sigma$. For $\widetilde{C}$, we have the following cases:

1. $\widetilde{C} = \mathbf{skip}$. Trivial.
2. $\widetilde{C} = \mathbf{linself}$. By the instrumented operational semantics.
3. $\widetilde{C}$ is $\mathbf{trylinself}$, or $\mathbf{lin}(E)$, or $\mathbf{trylin}(E)$ or $\mathbf{commit}(p)$. These cases are all similar to the previous one.

Inductive Step: $n = k + 1$. Thus there exist C_1 and σ_1 such that
$$(C, \sigma) \longrightarrow_{\mathsf{t}} (C_1, \sigma_1)\,, \text{ and } (C_1, \sigma_1) \longrightarrow_{\mathsf{t}}^k (\mathbf{skip}, \sigma')\,.$$
By case analysis of $(C, \sigma) \longrightarrow_{\mathsf{t}} (C_1, \sigma_1)$, which will be similar to Lemma 42, we have: there exist $\widetilde{C}_1$ and Δ_1 such that
$$(\widetilde{C}, (\sigma, \Delta)) \hookrightarrow_{\mathsf{t}} (\widetilde{C}_1, (\sigma_1, \Delta_1)), \text{ and } \mathsf{Er}(\widetilde{C}_1) = C_1.$$
By the induction hypothesis, we know there exists Δ' such that
$$(\widetilde{C}_1, (\sigma_1, \Delta_1)) \hookrightarrow_{\mathsf{t}}^* (\mathbf{skip}, (\sigma', \Delta')).$$
Thus we get the conclusion. □

Lemma 44 (From Concrete to Instrumented Steps: Abort). For any C, σ, Δ, t and $\widetilde{C}$, if

1. $(C, \sigma) \longrightarrow_{\mathsf{t}} \mathbf{abort}$,
2. $\mathsf{Er}(\widetilde{C}) = C$, where $C \neq \mathbf{E}[\,\mathbf{return}\ \_\,]$,

then $(\widetilde{C}, (\sigma, \Delta)) \hookrightarrow_{\mathsf{t}} \mathbf{abort}$.

Proof: By case analysis of $(C, \sigma) \longrightarrow\!\!\triangleright_{\mathsf{t}}$ **abort**. The proof is similar to Lemma 42. $\square$

Lemma 45 (From Instrumented to Speculative Steps).
For any $\widetilde{C}$, σ, Δ, $\widetilde{C}'$, σ', Δ' and t,
if $(\widetilde{C}, (\sigma, \Delta)) \hookrightarrow_{\mathsf{t}} (\widetilde{C}', (\sigma', \Delta'))$, then $\Delta \rightrightarrows \Delta'$.

Proof: By case analysis of $(\widetilde{C}, (\sigma, \Delta)) \hookrightarrow_{\mathsf{t}} (\widetilde{C}', (\sigma', \Delta'))$. The proof is similar to Lemma 42.
For example, if $\widetilde{C} = $ **linself**, we know $\Delta \rightarrow_{\mathsf{t}} \Delta'$, which we can prove implies $\Delta \rightrightarrows \Delta'$. $\square$

Lemma 46 (From Judgment Semantics to Simulation).
For any t, x, C, γ, R, G and p, if there exist I and $\widetilde{C}$ such that

$$R, G, I \models_{\mathsf{t}} \{\mathsf{t} \rightarrowtail (\gamma, x) * p\}\ \widetilde{C}\ \{\mathsf{t} \rightarrowtail (\mathbf{end}, \_) * (x = \_) * p\},$$

and $\mathsf{Er}(\widetilde{C}) = (C; \mathbf{noret})$, then $(x, C) \preceq^{\mathsf{t}}_{R;G;p} \gamma$.

Proof: We want to prove: for any n, σ and Δ,
if $(\sigma, \Delta) \models (\mathsf{t} \rightarrowtail (\gamma, n) * (x = n) * p)$, then

$$(C; \mathbf{noret}, \sigma) \preceq^{\mathsf{t}}_{R;G;p} \Delta\ .$$

We have $(\sigma, \Delta) \models (\mathsf{t} \rightarrowtail (\gamma, x) * p)$.
Then from the premise, we know the following are true (where $\mathcal{R} = [\![R * \mathsf{Id}]\!]$ and $\mathcal{G} = [\![G * \mathsf{True}]\!]$):

1. for any Σ', if $(\widetilde{C}, (\sigma, \Delta)) \overset{\mathcal{R}}{\hookrightarrow}{}^{*}_{\mathsf{t}} (\mathbf{skip}, \Sigma')$, then $\Sigma' \models (\mathsf{t} \rightarrowtail (\mathbf{end}, \_) * (x = \_) * p)$;
2. for any n, $(\widetilde{C}, (\sigma, \Delta), \mathcal{R})\ \mathsf{guar}^{n}_{\mathsf{t}}\ \mathcal{G}$.

Let $C_1 = (C; \mathbf{noret})$. Thus $\mathsf{Er}(\widetilde{C}) = C_1$. We prove

$$(C_1, \sigma) \preceq^{\mathsf{t}}_{R;G;p} \Delta$$

by its definition and co-induction. We have the following cases:

1. If $C_1 \neq \mathbf{E}[\,\mathbf{return}\ \_\,]$, then
   (a) for any C' and σ', if $(C_1, \sigma) \longrightarrow\!\!\triangleright_{\mathsf{t}} (C', \sigma')$,
   by Lemma 42, we know there exist $\widetilde{C}'$ and Δ' such that
   $(\widetilde{C}, (\sigma, \Delta)) \hookrightarrow_{\mathsf{t}} (\widetilde{C}', (\sigma', \Delta'))$, and $\mathsf{Er}(\widetilde{C}') = C'$.
   Then by Lemma 45, we know
   $$\Delta \rightrightarrows \Delta'\ .$$
   From premise 2, we know
   $$((\sigma, \Delta), (\sigma', \Delta')) \in \mathcal{G},$$
   thus $((\sigma, \Delta), (\sigma', \Delta')) \models (G * \mathsf{True})$.
   Finally, from the hypothesis, we know $(C', \sigma') \preceq^{\mathsf{t}}_{R;G;p} \Delta'$.
   (b) From premise 2, we know
   $$(\widetilde{C}, (\sigma, \Delta)) \not\hookrightarrow_{\mathsf{t}} \mathbf{abort}\,.$$
   By Lemma 44, we know $(C_1, \sigma) \not\longrightarrow\!\!\triangleright_{\mathsf{t}}$ **abort**.
2. If $C_1 = \mathbf{E}[\,\mathbf{return}\ E\,]$,
   then $\widetilde{C} = \mathbf{E}'[\,\mathbf{return}\ E\,]$ where $\mathsf{Er}(\mathbf{E}') = \mathbf{E}$.
   From premise 2, we know there exists n' such that $[\![E]\!]_{\sigma} = n'$ and
   $$\forall U.\ (U, \_) \in \Delta\ \Rightarrow\ U(\mathsf{t}) = (\mathbf{end}, n') \qquad \text{(C.38)}$$
   Also we have
   $$(\widetilde{C}, (\sigma, \Delta)) \hookrightarrow_{\mathsf{t}} (\mathbf{skip}, (\sigma, \Delta))$$
   Then from premise 1, we know
   $$(\sigma, \Delta) \models (\mathsf{t} \rightarrowtail (\mathbf{end}, \_) * (x = \_) * p)\,.$$
   Then from (C.38), we know
   $$(\sigma, \Delta) \models (\mathsf{t} \rightarrowtail (\mathbf{end}, n') * (x = \_) * p)\,.$$
3. For any σ' and Δ', if $((\sigma, \Delta), (\sigma', \Delta')) \models (R * \mathsf{Id})$, then $((\sigma, \Delta), (\sigma', \Delta')) \in \mathcal{R}$.
   By the hypothesis, we know $(C_1, \sigma') \preceq^{\mathsf{t}}_{R;G;p} \Delta'$.

Thus we get $(C; \mathbf{noret}, \sigma) \preceq^{\mathsf{t}}_{R;G;p} \Delta$, and finish the proof. $\square$

C.2.2 Soundness of Inference Rules

Theorem 40 (logic soundness *w.r.t.* the standard rely-guarantee-style semantics) is proved by induction over the derivation of the judgment $R, G, I \vdash_{\mathsf{t}} \{p\}\widetilde{C}\{q\}$. The whole proof consists of the soundness proof for each individual rules. Here we show the main lemmas used to prove the soundness of RET, FRAME, SPEC-CONJ, COMMIT, LINSELF, LINSELF-END, TRY and TRY-END.

The* RET *rule.

Lemma 47 (Ret-Sound).
$\models_{\mathsf{t}} \{\mathsf{t} \rightarrowtail (\mathbf{end}, E)\}\mathbf{E}[\,\mathbf{return}\ E\,]\{\mathsf{t} \rightarrowtail (\mathbf{end}, E)\}$.

Proof: For any σ and Δ such that $(\sigma, \Delta) \models (\mathsf{t} \rightarrowtail (\mathbf{end}, E))$, we know there exists n such that

$$\{\!\!\{E\}\!\!\}_{\sigma} = n, \ \text{ and } \ \Delta = \{(\{\mathsf{t} \rightsquigarrow (\mathbf{end}, n)\}, \emptyset)\}\,.$$

Thus we know $[\![E]\!]_{\sigma} = n$ and

$$\forall U.\ (U, \_) \in \Delta\ \Rightarrow\ U(\mathsf{t}) = (\mathbf{end}, n).$$

Then

$$(\mathbf{E}[\,\mathbf{return}\ E\,], (\sigma, \Delta)) \hookrightarrow_{\mathsf{t}} (\mathbf{skip}, (\sigma, \Delta))\,.$$

By Definition 38, we get the conclusion. $\square$

The* LINSELF *rule. We first prove the following useful lemma:

Lemma 48. For any Δ_1, Δ_1', Δ_2, Δ_2', t, γ, n and n', if

1. $(\Delta_2, n) \overset{\gamma}{\rightarrow} (\Delta_2', n')$,
2. $\Delta_1 = \{(\{\mathsf{t} \rightsquigarrow (\gamma, n)\}, \emptyset)\}$, $\ \Delta_1' = \{(\{\mathsf{t} \rightsquigarrow (\mathbf{end}, n')\}, \emptyset)\}$,

then $(\Delta_1 * \Delta_2) \rightarrow_{\mathsf{t}} (\Delta_1' * \Delta_2')$.

Lemma 49 (Linself-Sound). If $[E_1, p]\gamma[E_2, q]$,
then $\models_{\mathsf{t}} \{\mathsf{t} \rightarrowtail (\gamma, E_1) * p\}\mathbf{linself}\{\mathsf{t} \rightarrowtail (\mathbf{end}, E_2) * q\}$.

Proof: For any σ and Δ such that $(\sigma, \Delta) \models (\mathsf{t} \rightarrowtail (\gamma, E_1) * p)$, we know there exist n, σ_1, σ_2, Δ_1 and Δ_2 such that

$$\{\!\!\{E_1\}\!\!\}_{\sigma_1} = n,\ \ \Delta_1 = \{(\{\mathsf{t} \rightsquigarrow (\gamma, n)\}, \emptyset)\},$$
$$(\sigma_2, \Delta_2) \models p,\ \ \sigma = \sigma_1 \uplus \sigma_2 \ \text{ and } \ \Delta = \Delta_1 * \Delta_2\,.$$

From $[E_1, p]\gamma[E_2, q]$, we know: there exist σ_1', σ_2', Δ_2' and n' such that

$$\{\!\!\{E_2\}\!\!\}_{\sigma_1'} = n',\ \ (\Delta_2, n) \overset{\gamma}{\rightarrow} (\Delta_2', n'),$$
$$(\sigma_2', \Delta_2') \models q \ \text{ and } \ \sigma_1 \uplus \sigma_2 = \sigma_1' \uplus \sigma_2'\,.$$

By Lemma 48, we know

$$\Delta \rightarrow_{\mathsf{t}} (\Delta_1' * \Delta_2'),\ \text{ where } \ \Delta_1' = \{(\{\mathsf{t} \rightsquigarrow (\mathbf{end}, n')\}, \emptyset)\}\,.$$

Thus

$$(\mathbf{linself}, (\sigma, \Delta)) \hookrightarrow_{\mathsf{t}} (\mathbf{skip}, (\sigma, \Delta_1' * \Delta_2'))\,,$$

where

$$(\sigma, \Delta_1' * \Delta_2') \models (\mathsf{t} \rightarrowtail (\mathbf{end}, E_2) * q)\,.$$

By Definition 38, we get the conclusion. $\square$

The* LINSELF-END *rule.

Lemma 50 (Linself-End-Sound).
$\models_{\mathsf{t}} \{\mathsf{t} \rightarrowtail (\mathbf{end}, E)\}\mathbf{linself}\{\mathsf{t} \rightarrowtail (\mathbf{end}, E)\}\,.$

Proof: For any σ and Δ such that $(\sigma, \Delta) \models (\mathsf{t} \rightarrowtail (\mathbf{end}, E))$, we know there exists n such that

$$\{\!\!\{E\}\!\!\}_{\sigma} = n, \ \text{ and } \ \Delta = \{(\{\mathsf{t} \rightsquigarrow (\mathbf{end}, n)\}, \emptyset)\}\,.$$

Thus $\Delta \rightarrow_{\mathsf{t}} \Delta$. Then

$$(\textbf{linself}, (\sigma, \Delta)) \hookrightarrow_{\mathsf{t}} (\textbf{skip}, (\sigma, \Delta))\,.$$

By Definition 38, we get the conclusion. □

The* TRY *rule.

Lemma 51 (Try-Sound). If $[E_1, p]\gamma[E_2, q]$, then $\models_{\mathsf{t}} \{E \rightarrowtail (\gamma, E_1) * p\}\textbf{trylin}(E)\{(E \rightarrowtail (\gamma, E_1) * p) \oplus (E \rightarrowtail (\textbf{end}, E_2) * q)\}$.

Proof: For any σ and Δ such that $(\sigma, \Delta) \models (E \rightarrowtail (\gamma, E_1) * p)$, we know there exist t', n, σ_{11}, σ_{12}, σ_2, Δ_1 and Δ_2 such that

$$\begin{array}{c}\{\!\!\{E\}\!\!\}_{\sigma_{11}} = \mathsf{t}',\ \ \{\!\!\{E_1\}\!\!\}_{\sigma_{12}} = n,\ \ \Delta_1 = \{(\{\mathsf{t}' \rightsquigarrow (\gamma, n)\}, \emptyset)\},\\ (\sigma_2, \Delta_2) \models p,\ \ \sigma = \sigma_{11} \uplus \sigma_{12} \uplus \sigma_2 \ \ \text{and} \ \ \Delta = \Delta_1 * \Delta_2\,.\end{array}$$

From $[E_1, p]\gamma[E_2, q]$, we know: there exist σ'_{12}, σ'_2, Δ'_2 and n' such that

$$\begin{array}{c}\{\!\!\{E_2\}\!\!\}_{\sigma'_{12}} = n',\ \ (\Delta_2, n) \xrightarrow{\gamma} (\Delta'_2, n'),\\ (\sigma'_2, \Delta'_2) \models q \ \ \text{and} \ \ \sigma_{12} \uplus \sigma_2 = \sigma'_{12} \uplus \sigma'_2\,.\end{array}$$

By Lemma 48, we know

$$\Delta \rightarrow_{\mathsf{t}'} (\Delta'_1 * \Delta'_2), \ \ \text{where} \ \ \Delta'_1 = \{(\{\mathsf{t}' \rightsquigarrow (\textbf{end}, n')\}, \emptyset)\}\,.$$

Thus

$$(\textbf{trylin}(E), (\sigma, \Delta)) \hookrightarrow_{\mathsf{t}} (\textbf{skip}, (\sigma, (\Delta'_1 * \Delta'_2) \cup \Delta))\,,$$

where

$$(\sigma, \Delta'_1 * \Delta'_2) \models (E \rightarrowtail (\textbf{end}, E_2) * q)\,,$$

thus

$$(\sigma, (\Delta'_1 * \Delta'_2) \cup \Delta) \models (E \rightarrowtail (\gamma, E_1) * p) \oplus (E \rightarrowtail (\textbf{end}, E_2) * q)\,.$$

By Definition 38, we get the conclusion. □

The* TRY-END *rule.

Lemma 52 (Try-End-Sound).
$\models_{\mathsf{t}} \{E \rightarrowtail (\textbf{end}, E')\}\textbf{trylin}(E)\{E \rightarrowtail (\textbf{end}, E')\}$.

Proof: For any σ and Δ such that $(\sigma, \Delta) \models (E \rightarrowtail (\textbf{end}, E'))$, we know there exist σ_1, σ_2, t' and n such that

$$\begin{array}{c}\{\!\!\{E\}\!\!\}_{\sigma_1} = \mathsf{t}',\ \ \{\!\!\{E\}\!\!\}_{\sigma_2} = n,\\ \sigma = \sigma_1 \uplus \sigma_2 \ \ \text{and} \ \ \Delta = \{(\{\mathsf{t}' \rightsquigarrow (\textbf{end}, n)\}, \emptyset)\}\,.\end{array}$$

Thus $\Delta \rightarrow_{\mathsf{t}'} \Delta$. Then

$$(\textbf{trylin}(E), (\sigma, \Delta)) \hookrightarrow_{\mathsf{t}} (\textbf{skip}, (\sigma, \Delta))\,.$$

By Definition 38, we get the conclusion. □

The* COMMIT *rule. First we prove some properties on $(\sigma, \Delta)|_p = (\sigma', \Delta')$. We use δ as a shorthand for (U, θ).

Lemma 53. 1. If $\Delta|_D = \Delta'$, then for any $\delta \in \Delta$, there exist δ' and δ'' such that $\delta = \delta' \uplus \delta''$ and $\delta' \in \Delta'$.
2. If $\Delta|_{dom(\Delta')} \cap \Delta' = \emptyset$, then for any $\delta \in \Delta$, there does not exist δ' or δ'' such that $\delta = \delta' \uplus \delta''$ and $\delta' \in \Delta'$.

Proof: 1. Since $\Delta|_D = \Delta' \neq \emptyset$, we know

$$\Delta|_D = \{\delta' \mid dom(\delta') = D \wedge \exists \delta''.\ \delta' \uplus \delta'' \in \Delta\} \neq \emptyset$$

Thus there exist δ_0, δ'_0 and δ''_0 such that $\delta_0 = \delta'_0 \uplus \delta''_0 \in \Delta$ and $dom(\delta'_0) = D$. Then we know

$$D \subseteq dom(\Delta)$$

Thus for any $\delta \in \Delta$, there exist δ' and δ'' such that

$$\delta = \delta' \uplus \delta''\,, \quad dom(\delta') = D$$

Thus $\delta' \in \Delta|_D$, and hence $\delta' \in \Delta'$.

2. Since $\Delta|_{dom(\Delta')} \cap \Delta' = \emptyset$, we immediately know

$$\neg(\exists \delta', \delta''.\ (\delta' \uplus \delta'' \in \Delta) \wedge (\delta' \in \Delta'))$$

Thus we get the conclusion.

□

From Lemma 53, we know:

If $\Delta_1|_{dom(\Delta)} = \Delta$ and $\Delta_2|_{dom(\Delta)} \cap \Delta = \emptyset$, then $\Delta_1 \cap \Delta_2 = \emptyset$.

Below we prove that $(\sigma, \Delta)|_p = (\_, \Delta')$ is deterministic when $\mathsf{SpecExact}(p)$.

Lemma 54. If $\mathsf{SpecExact}(p)$, and both $(\sigma, \Delta)|_p = (\sigma'_1, \Delta'_1)$ and $(\sigma, \Delta)|_p = (\sigma'_2, \Delta'_2)$ hold, then $\Delta'_1 = \Delta'_2$.

Proof: We know there exist Δ_p, σ''_1, σ''_2, Δ''_1 and Δ''_2 such that

$$\begin{array}{l}(\sigma = \sigma'_1 \uplus \sigma''_1) \wedge (\Delta = \Delta'_1 \uplus \Delta''_1) \wedge ((\sigma'_1, \Delta_p) \models p)\\ \wedge (\Delta'_1|_{dom(\Delta_p)} = \Delta_p) \wedge (\Delta''_1|_{dom(\Delta_p)} \cap \Delta_p = \emptyset)\,,\\ (\sigma = \sigma'_2 \uplus \sigma''_2) \wedge (\Delta = \Delta'_2 \uplus \Delta''_2) \wedge ((\sigma'_2, \Delta_p) \models p)\\ \wedge (\Delta'_2|_{dom(\Delta_p)} = \Delta_p) \wedge (\Delta''_2|_{dom(\Delta_p)} \cap \Delta_p = \emptyset)\,.\end{array}$$

Since $\Delta'_1|_{dom(\Delta_p)} = \Delta_p$ and $\Delta''_2|_{dom(\Delta_p)} \cap \Delta_p = \emptyset$, by Lemma 53, we know $\Delta'_1 \cap \Delta''_2 = \emptyset$. Similarly we know $\Delta''_1 \cap \Delta'_2 = \emptyset$. Since $\Delta'_1 \uplus \Delta''_1 = \Delta'_2 \uplus \Delta''_2$, we know $\Delta'_1 = \Delta'_2$. □

Lemma 55 (Commit-Sound). If $\mathsf{SpecExact}(p)$ and $p' \Rightarrow p$, then $\models_{\mathsf{t}} \{p' \oplus \mathsf{true}\}\textbf{commit}(p)\{p'\}$.

Proof: For any σ and Δ such that $(\sigma, \Delta) \models p' \oplus \mathsf{true}$, we know there exist Δ' and Δ'' such that

$$(\sigma, \Delta') \models p'\,, \ \ \Delta = \Delta' \uplus \Delta''\,.$$

Thus $(\sigma, \Delta') \models p$. We know $\Delta'|_{dom(\Delta')} = \Delta'$ and $\Delta''|_{dom(\Delta')} = \Delta''$. Then $\Delta''|_{dom(\Delta')} \cap \Delta' = \emptyset$. Thus

$$(\sigma, \Delta)|_p = (\sigma, \Delta')\,.$$

Thus by the operational semantics, we know

$$(\textbf{commit}(p), (\sigma, \Delta)) \not\hookrightarrow_{\mathsf{t}} \textbf{abort}\,.$$

On the other hand, for any Δ_1 such that $(\textbf{commit}(p), (\sigma, \Delta)) \hookrightarrow_{\mathsf{t}} (\textbf{skip}, (\sigma, \Delta_1))$, we know $(\sigma, \Delta)|_p = (\_, \Delta_1)$. By Lemma 54, we know $\Delta_1 = \Delta'$. Thus $(\sigma, \Delta_1) \models p'$ and we are done. □

Lemma 56 (Commit-Spec-Conj-Sound). If $\models_{\mathsf{t}} \{p_1\}\textbf{commit}(p)\{q\}$ and $p_2 \nmid p$, then $\models_{\mathsf{t}} \{p_1 \oplus p_2\}\textbf{commit}(p)\{q\}$.

Proof: For any σ and Δ such that $(\sigma, \Delta) \models p_1 \oplus p_2$, we know there exist Δ_1 and Δ_2 such that

$$(\sigma, \Delta_1) \models p_1\,, \ \ (\sigma, \Delta_2) \models p_2\,, \ \ \Delta = \Delta_1 \cup \Delta_2\,.$$

From $\models_{\mathsf{t}} \{p_1\}\textbf{commit}(p)\{q\}$, we know

(1) for any Δ'_1, if $(\textbf{commit}(p), (\sigma, \Delta_1)) \hookrightarrow^*_{\mathsf{t}} (\textbf{skip}, (\sigma, \Delta'_1))$, then $(\sigma, \Delta'_1) \models q$;
(2) $(\textbf{commit}(p), (\sigma, \Delta_1)) \not\hookrightarrow^*_{\mathsf{t}} \textbf{abort}$.

From (2), we know $\mathsf{SpecExact}(p)$ and there exist σ' and Δ'_1 such that $(\sigma, \Delta_1)|_p = (\sigma', \Delta'_1)$. Thus there exist σ'', Δ''_1 and Δ_p such that

$$\begin{array}{l}\sigma = \sigma' \uplus \sigma''\,, \ \ \Delta_1 = \Delta'_1 \uplus \Delta''_1\,, \ \ (\sigma', \Delta_p) \models p\,,\\ \Delta'_1|_{dom(\Delta_p)} = \Delta_p\,, \ \ \Delta''_1|_{dom(\Delta_p)} \cap \Delta_p = \emptyset\,.\end{array}$$

Since $(\sigma, \Delta_2) \models p_2$ and $p_2 \nmid p$, we know

$$\Delta_2|_{dom(\Delta_p)} \cap \Delta_p = \emptyset\,.$$

Thus $(\Delta_1'' \cup \Delta_2)|_{dom(\Delta_p)} \cap \Delta_p = \emptyset$. Then by Lemma 53, we know $\Delta_1' \cap (\Delta_1'' \cup \Delta_2) = \emptyset$. Thus we have

$$\Delta = \Delta_1' \uplus (\Delta_1'' \cup \Delta_2)\,,\ \ \Delta_1'|_{dom(\Delta_p)} = \Delta_p\,,\\ (\Delta_1'' \cup \Delta_2)|_{dom(\Delta_p)} \cap \Delta_p = \emptyset\,.$$

Thus $(\sigma, \Delta)|_p = (\sigma', \Delta_1')$ holds. Then, by the operational semantics, we know

$$(\textbf{commit}(p), (\sigma, \Delta)) \not\hookrightarrow_{\rm t} \textbf{abort}\,.$$

On the other hand, for any Δ' such that $(\textbf{commit}(p), (\sigma, \Delta)) \hookrightarrow_{\rm t} (\textbf{skip}, (\sigma, \Delta'))$, we know $(\sigma, \Delta)|_p = (\_, \Delta')$. By Lemma 54, we know $\Delta_1' = \Delta'$. Since $(\sigma, \Delta_1)|_p = (\sigma', \Delta_1')$, we know $(\textbf{commit}(p), (\sigma, \Delta_1)) \hookrightarrow_{\rm t} (\textbf{skip}, (\sigma, \Delta_1'))$. From (1), we know $(\sigma, \Delta_1') \models q$ and we are done. □

Lemma 57 (Multi-Commit-Sound). If $\models_{\rm t} \{p\}\textbf{commit}(p_1)\{q_1\}$, $\models_{\rm t} \{p\}\textbf{commit}(p_2)\{q_2\}$, $\mathsf{Exact}(p_1)$, $\mathsf{Exact}(p_2)$ and $p_1 \oplus p_2$ is satisfiable, then $\models_{\rm t} \{p\}\textbf{commit}(p_1 \oplus p_2)\{q_1 \oplus q_2\}$.

Proof: For any σ and Δ such that $(\sigma, \Delta) \models p$, from the premises, we know

(1) for any Δ_1', if $(\textbf{commit}(p_1), (\sigma, \Delta)) \hookrightarrow_{\rm t}^* (\textbf{skip}, (\sigma, \Delta_1'))$, then $(\sigma, \Delta_1') \models q_1$;
(2) $(\textbf{commit}(p_1), (\sigma, \Delta)) \not\hookrightarrow_{\rm t}^* \textbf{abort}$;
(3) for any Δ_2', if $(\textbf{commit}(p_2), (\sigma, \Delta)) \hookrightarrow_{\rm t}^* (\textbf{skip}, (\sigma, \Delta_2'))$, then $(\sigma, \Delta_2') \models q_1$;
(4) $(\textbf{commit}(p_2), (\sigma, \Delta)) \not\hookrightarrow_{\rm t}^* \textbf{abort}$.

Since $\mathsf{Exact}(p_1)$ and $\mathsf{Exact}(p_2)$, we know $\mathsf{Exact}(p_1 \oplus p_2)$, thus $\mathsf{SpecExact}(p_1 \oplus p_2)$. From (2) and (4), we know there exist σ_1', Δ_1', σ_2' and Δ_2' such that $(\sigma, \Delta)|_{p_1} = (\sigma_1', \Delta_1')$ and $(\sigma, \Delta)|_{p_2} = (\sigma_2', \Delta_2')$. Thus there exist σ_1'', Δ_1'', Δ_{p1}, σ_2'', Δ_2'' and Δ_{p2} such that

$$\sigma = \sigma_1' \uplus \sigma_1''\,,\ \ \Delta = \Delta_1' \uplus \Delta_1''\,,\ \ (\sigma_1', \Delta_{p1}) \models p_1\,,\\ \Delta_1'|_{dom(\Delta_{p1})} = \Delta_{p1}\,,\ \ \Delta_1''|_{dom(\Delta_{p1})} \cap \Delta_{p1} = \emptyset\,;\\ \sigma = \sigma_2' \uplus \sigma_2''\,,\ \ \Delta = \Delta_2' \uplus \Delta_2''\,,\ \ (\sigma_2', \Delta_{p2}) \models p_2\,,\\ \Delta_2'|_{dom(\Delta_{p2})} = \Delta_{p2}\,,\ \ \Delta_2''|_{dom(\Delta_{p2})} \cap \Delta_{p2} = \emptyset\,.$$

Since $p_1 \oplus p_2$ is satisfiable, we know there exist σ' and Δ_p such that $(\sigma', \Delta_p) \models p_1 \oplus p_2$. Since $\mathsf{Exact}(p_1)$ and $\mathsf{Exact}(p_2)$, we know $\sigma' = \sigma_1' = \sigma_2'$ and $\Delta_p = \Delta_{p1} \cup \Delta_{p2}$. Thus $dom(\Delta_p) = dom(\Delta_{p1}) = dom(\Delta_{p2})$.

Below we prove $(\sigma, \Delta)|_{p_1 \oplus p_2} = (\sigma', \Delta_1' \cup \Delta_2')$. We know $\sigma = \sigma' \uplus \sigma_1''$, and there exists Δ'' such that $\Delta = (\Delta_1' \cup \Delta_2') \uplus \Delta''$. Thus $\Delta'' \subseteq \Delta_1''$ and $\Delta'' \subseteq \Delta_2''$. Then

$$\begin{aligned}&(\Delta_1' \cup \Delta_2')|_{dom(\Delta_p)} = \Delta_1'|_{dom(\Delta_p)} \cup \Delta_2'|_{dom(\Delta_p)}\\ &\qquad\qquad\qquad\qquad = \Delta_{p1} \cup \Delta_{p2} = \Delta_p\,,\\ &\Delta''|_{dom(\Delta_p)} \cap \Delta_p\\ &= (\Delta''|_{dom(\Delta_p)} \cap \Delta_{p1}) \cup (\Delta''|_{dom(\Delta_p)} \cap \Delta_{p2})\\ &\subseteq (\Delta_1''|_{dom(\Delta_{p1})} \cap \Delta_{p1}) \cup (\Delta_2''|_{dom(\Delta_{p2})} \cap \Delta_{p2}) = \emptyset\end{aligned}$$

Thus we get $(\sigma, \Delta)|_{p_1 \oplus p_2} = (\sigma', \Delta_1' \cup \Delta_2')$. By the operational semantics, we know

$$(\textbf{commit}(p), (\sigma, \Delta)) \not\hookrightarrow_{\rm t} \textbf{abort}\,.$$

On the other hand, for any Δ' such that $(\textbf{commit}(p), (\sigma, \Delta)) \hookrightarrow_{\rm t} (\textbf{skip}, (\sigma, \Delta'))$, we know $(\sigma, \Delta)|_p = (\_, \Delta')$. By Lemma 54, we know $\Delta_1' \cup \Delta_2' = \Delta'$. From (1) and (3), we know $(\sigma, \Delta_1') \models q_1$ and $(\sigma, \Delta_2') \models q_2$. Thus $(\sigma, \Delta') \models q_1 \oplus q_2$ and we are done. □

The* FRAME *rule and locality.

Definition 58 (Locality). If $(\widetilde{C}, \Sigma_1) \not\hookrightarrow_{\rm t}^* \textbf{abort}$, then for all Σ_2 and $\Sigma = \Sigma_1 * \Sigma_2$,

1. (Safety property) $(\widetilde{C}, \Sigma) \not\hookrightarrow_{\rm t}^* \textbf{abort}$;
2. (Frame property) for all $\widetilde{C}'$ and Σ', if $(\widetilde{C}, \Sigma) \hookrightarrow_{\rm t}^* (\widetilde{C}', \Sigma')$, then there exists Σ_1' such that $\Sigma' = \Sigma_1' * \Sigma_2$ and $(\widetilde{C}, \Sigma_1) \hookrightarrow_{\rm t}^* (\widetilde{C}', \Sigma_1')$.

We prove the frame property of $\textbf{commit}(p)$ below:

Lemma 59. If

1. $(\textbf{commit}(p), (\sigma, \Delta)) \hookrightarrow_{\rm t} (\textbf{skip}, (\sigma, \Delta'))$,
2. $\sigma = \sigma_1 \uplus \sigma_2,\ \Delta = \Delta_1 * \Delta_2$,
3. $(\textbf{commit}(p), (\sigma_1, \Delta_1)) \not\hookrightarrow_{\rm t} \textbf{abort}$,

then there exists Δ_1' such that

1. $\Delta' = \Delta_1' * \Delta_2$, and
2. $(\textbf{commit}(p), (\sigma_1, \Delta_1)) \hookrightarrow_{\rm t} (\textbf{skip}, (\sigma_1, \Delta_1'))$.

Proof: From premise 1, we know $\mathsf{SpecExact}(p)$ and there exist σ_p, Δ_p and D such that $(\sigma_p, \Delta_p) \models p$ and $dom(\Delta_p) = D$, and there exists Δ'' such that

$$\Delta = \Delta' \uplus \Delta''\,,\ \ \sigma_p \subseteq \sigma\,,\ \ \Delta'|_D = \Delta_p\,,\ \ \Delta''|_D \cap \Delta_p = \emptyset$$

From premise 3 and $\mathsf{SpecExact}(p)$, we know there exist σ_p', Δ_1' and Δ_1'' such that $(\sigma_p', \Delta_p) \models p$ and

$$\Delta_1 = \Delta_1' \uplus \Delta_1''\,,\ \ \sigma_p' \subseteq \sigma_1\,,\ \ \Delta_1'|_D = \Delta_p\,,\ \ \Delta_1''|_D \cap \Delta_p = \emptyset$$

Below we prove $\Delta' = \Delta_1' * \Delta_2$.

1. We prove $\Delta_1' \sharp \Delta_2$.
   Since $\Delta_1 \sharp \Delta_2$, and $\Delta_1' \subseteq \Delta_1$, we know $\Delta_1' \sharp \Delta_2$.
2. We prove $\Delta_1' * \Delta_2 \subseteq \Delta'$.
   For any δ such that $\delta \in \Delta_1' * \Delta_2$, we know there exist δ_1' and δ_2 such that
   $$\delta_1' \in \Delta_1'\,,\quad \delta_2 \in \Delta_2\,,\quad \delta = \delta_1' \uplus \delta_2$$
   From $\delta_1' \in \Delta_1'$ and $\Delta_1'|_D = \Delta_p$, by the above Lemma 53, we know there exist δ_p and δ' such that
   $$\delta_1' = \delta_p \uplus \delta'\,,\quad \delta_p \in \Delta_p$$
   Thus we know
   $$\delta = \delta_p \uplus (\delta' \uplus \delta_2)$$
   Since $\Delta''|_{dom(\Delta_p)} \cap \Delta_p = \emptyset$, by the above Lemma 53, we know $\delta \notin \Delta''$. Since $\delta \in \Delta_1' * \Delta_2 \subseteq \Delta_1 * \Delta_2 = \Delta$, we know $\delta \in \Delta'$.
3. We prove $\Delta' \subseteq \Delta_1' * \Delta_2$.
   For any δ such that $\delta \in \Delta'$, by the above Lemma 53, we know there exist δ_p and δ' such that
   $$\delta = \delta_p \uplus \delta'\,,\quad \delta_p \in \Delta_p$$
   Since $\Delta_1'|_D = \Delta_p$, by the above Lemma 53, we know $dom(\Delta_p) \subseteq dom(\Delta_1')$. Since $\Delta_1' \sharp \Delta_2$, we know $\Delta_p \sharp \Delta_2$. Thus $dom(\delta_p) \cap dom(\Delta_2) = \emptyset$. Since $dom(\Delta_2) \subseteq dom(\delta) = dom(\delta_p) \uplus dom(\delta')$, we know $dom(\Delta_2) \subseteq dom(\delta')$. Thus there exist δ_1' and δ_2' such that
   $$\delta' = \delta_1' \uplus \delta_2'\,,\quad dom(\delta_2') = dom(\Delta_2)$$
   On the other hand, since $\delta \in \Delta' \subseteq \Delta = \Delta_1 * \Delta_2$, we know there exist δ_1 and δ_2 such that
   $$\delta = \delta_1 \uplus \delta_2\,,\ \ \delta_1 \in \Delta_1\,,\ \ \delta_2 \in \Delta_2\,,\ \ dom(\delta_2) = dom(\Delta_2)$$
   Since $\delta = \delta_p \uplus \delta_1' \uplus \delta_2' = \delta_1 \uplus \delta_2$ and $dom(\delta_2') = dom(\delta_2)$, we know $\delta_2' = \delta_2$ and $\delta_p \uplus \delta_1' = \delta_1$. Since $\Delta_1''|_{dom(\Delta_p)} \cap \Delta_p = \emptyset$, by the above Lemma 53, we know $\delta_1 \notin \Delta_1''$. Since $\delta_1 \in \Delta_1$, we know $\delta_1 \in \Delta_1'$. Thus, $\delta \in \Delta_1' * \Delta_2$.

Thus we are done. □

The* SPEC-CONJ *rule. We first prove Lemmas 62 and 63 below, which say speculations can be split during executions.

Lemma 60. If $\Delta \rightarrow_{\mathsf{t}} \Delta'$ and $\Delta = \Delta_1 \cup \Delta_2$, then there exist Δ'_1 and Δ'_2 such that $\Delta_1 \rightarrow_{\mathsf{t}} \Delta'_1$, $\Delta_2 \rightarrow_{\mathsf{t}} \Delta'_2$ and $\Delta' = \Delta'_1 \cup \Delta'_2$.

Lemma 61. If

1. $\mathsf{SpecExact}(p)$,
2. $\Delta = \Delta_1 \cup \Delta_2$,
3. $(\sigma, \Delta_1)|_{dom(p)} = (\sigma'_1, \Delta'_1)$, $(\sigma, \Delta_2)|_{dom(p)} = (\sigma'_2, \Delta'_2)$,

then $(\sigma, \Delta)|_{dom(p)} = (\_, \Delta'_1 \cup \Delta'_2)$.

Proof: We know there exist σ''_1, Δ''_1, σ''_2, Δ''_2 and Δ_p such that

$$\begin{array}{l}\sigma = \sigma'_1 \uplus \sigma''_1\,,\ \ \Delta_1 = \Delta'_1 \uplus \Delta''_1\,,\ \ (\sigma'_1, \Delta_p) \models p\,,\\ \Delta'_1|_{dom(\Delta_p)} = \Delta_p\,,\ \ \Delta''_1|_{dom(\Delta_p)} \cap \Delta_p = \emptyset\,;\\ \sigma = \sigma'_2 \uplus \sigma''_2\,,\ \ \Delta_2 = \Delta'_2 \uplus \Delta''_2\,,\ \ (\sigma'_2, \Delta_p) \models p\,,\\ \Delta'_2|_{dom(\Delta_p)} = \Delta_p\,,\ \ \Delta''_2|_{dom(\Delta_p)} \cap \Delta_p = \emptyset\,.\end{array}$$

Thus

$$\begin{array}{l}\Delta = \Delta_1 \cup \Delta_2\\ = (\Delta'_1 \uplus \Delta''_1) \cup (\Delta'_2 \uplus \Delta''_2)\\ = (\Delta'_1 \cup \Delta'_2) \cup (\Delta''_1 \cup \Delta''_2)\,,\\ (\Delta'_1 \cup \Delta'_2)|_{dom(\Delta_p)}\\ = \Delta'_1|_{dom(\Delta_p)} \cup \Delta'_2|_{dom(\Delta_p)} = \Delta_p\,,\\ (\Delta''_1 \cup \Delta''_2)|_{dom(\Delta_p)} \cap \Delta_p\\ = (\Delta''_1|_{dom(\Delta_p)} \cap \Delta_p) \cup (\Delta''_2|_{dom(\Delta_p)} \cap \Delta_p) = \emptyset\,.\end{array}$$

Thus $(\sigma, \Delta)|_{dom(p)} = (\sigma'_1, \Delta'_1 \cup \Delta'_2)$. $\square$

Lemma 62. For any $\widetilde{C}$, $\widetilde{C}'$, σ, Δ, σ', Δ', Δ_1, Δ_2 and t, if

1. $(\widetilde{C}, (\sigma, \Delta)) \hookrightarrow_{\mathsf{t}} (\widetilde{C}', (\sigma', \Delta'))$,
2. $\Delta = \Delta_1 \cup \Delta_2$,
3. $(\widetilde{C}, (\sigma, \Delta_1)) \not\hookrightarrow_{\mathsf{t}}$ **abort**, $(\widetilde{C}, (\sigma, \Delta_2)) \not\hookrightarrow_{\mathsf{t}}$ **abort**,

then there exist Δ'_1 and Δ'_2 such that

1. $\Delta' = \Delta'_1 \cup \Delta'_2$,
2. $(\widetilde{C}, (\sigma, \Delta_1)) \hookrightarrow_{\mathsf{t}} (\widetilde{C}', (\sigma', \Delta'_1))$, and
3. $(\widetilde{C}, (\sigma, \Delta_2)) \hookrightarrow_{\mathsf{t}} (\widetilde{C}', (\sigma', \Delta'_2))$.

Proof: By case analysis over $(\widetilde{C}, (\sigma, \Delta)) \hookrightarrow_{\mathsf{t}} (\widetilde{C}', (\sigma', \Delta'))$.

1. $\widetilde{C}$ is $\mathbf{E}[\,c\,]$.
   Thus $(\mathbf{E}[\,c\,], \sigma) \longrightarrow_{\mathsf{t}} (\mathbf{E}[\,\mathbf{skip}\,], \sigma')$ and $\Delta = \Delta'$.
   Let $\Delta'_1 = \Delta_1$ and $\Delta'_2 = \Delta_2$, then we can get the conclusion.
2. $\widetilde{C}$ is $\mathbf{E}[\,\mathbf{return}\ E\,]$.
   Thus $(\mathbf{E}[\,\mathbf{return}\ E\,], (\sigma, \Delta)) \hookrightarrow_{\mathsf{t}} (\mathbf{skip}, (\sigma, \Delta))$, and
   $$[\![E]\!]_\sigma = n \ \text{ and } \ \forall U.\ (U, \_) \in \Delta \ \Rightarrow\ U(\mathsf{t}) = (\mathbf{end}, n)\,.$$
   Let $\Delta'_1 = \Delta_1$ and $\Delta'_2 = \Delta_2$, then we can get the conclusion.
3. $\widetilde{C}$ is $\mathbf{E}[\,\mathbf{linself}\,]$.
   Thus $(\mathbf{E}[\,\mathbf{linself}\,], (\sigma, \Delta)) \hookrightarrow_{\mathsf{t}} (\mathbf{E}[\,\mathbf{skip}\,], (\sigma, \Delta'))$, and
   $$\Delta \rightarrow_{\mathsf{t}} \Delta'\,.$$
   By Lemma 60, we know: there exist Δ'_1 and Δ'_2 such that $\Delta_1 \rightarrow_{\mathsf{t}} \Delta'_1$, $\Delta_2 \rightarrow_{\mathsf{t}} \Delta'_2$ and $\Delta' = \Delta'_1 \cup \Delta'_2$.
   Then we can get the conclusion.
4. $\widetilde{C}$ is $\mathbf{E}[\,\mathbf{trylin}(E)\,]$.
   Thus $(\mathbf{E}[\,\mathbf{trylin}(E)\,], (\sigma, \Delta)) \hookrightarrow_{\mathsf{t}} (\mathbf{E}[\,\mathbf{skip}\,], (\sigma, \Delta \cup \Delta'))$, and there exists t' such that
   $$[\![E]\!]_\sigma = \mathsf{t}' \ \text{ and } \ \Delta \rightarrow_{\mathsf{t}'} \Delta'\,.$$
   By Lemma 60, we know: there exist Δ'_1 and Δ'_2 such that $\Delta_1 \rightarrow_{\mathsf{t}'} \Delta'_1$, $\Delta_2 \rightarrow_{\mathsf{t}'} \Delta'_2$ and $\Delta' = \Delta'_1 \cup \Delta'_2$.
   Then we can get the conclusion.
5. $\widetilde{C}$ is $\mathbf{E}[\,\mathbf{commit}(p)\,]$. Below we prove:
   If $(\mathbf{commit}(p), (\sigma, \Delta)) \hookrightarrow_{\mathsf{t}} (\mathbf{skip}, (\sigma, \Delta'))$,
   $\Delta = \Delta_1 \cup \Delta_2$,
   $(\mathbf{commit}(p), (\sigma, \Delta_1)) \hookrightarrow_{\mathsf{t}} (\mathbf{skip}, (\sigma, \Delta'_1))$, and
   $(\mathbf{commit}(p), (\sigma, \Delta_2)) \hookrightarrow_{\mathsf{t}} (\mathbf{skip}, (\sigma, \Delta'_2))$,
   then $\Delta' = \Delta'_1 \cup \Delta'_2$.

   We know $\mathsf{SpecExact}(p)$, $(\sigma, \Delta_1)|_{dom(p)} = (\_, \Delta'_1)$, $(\sigma, \Delta_2)|_{dom(p)} = (\_, \Delta'_2)$ and $(\sigma, \Delta)|_{dom(p)} = (\_, \Delta')$. By Lemma 61, we know $(\sigma, \Delta)|_{dom(p)} = (\_, \Delta'_1 \cup \Delta'_2)$. By Lemma 54, we know $\Delta' = \Delta'_1 \cup \Delta'_2$.
6. Other cases are similar.

So we get the conclusion. $\square$

Lemma 63. For any $\widetilde{C}$, σ, Δ, Δ_1, Δ_2 and t, if

1. $(\widetilde{C}, (\sigma, \Delta_1)) \not\hookrightarrow_{\mathsf{t}}$ **abort**,
2. $(\widetilde{C}, (\sigma, \Delta_2)) \not\hookrightarrow_{\mathsf{t}}$ **abort**,
3. $\Delta = \Delta_1 \cup \Delta_2$,

then $(\widetilde{C}, (\sigma, \Delta)) \not\hookrightarrow_{\mathsf{t}}$ **abort**.

Proof: By case analysis over $\widetilde{C}$.

1. $\widetilde{C}$ is $\mathbf{E}[\,c\,]$.
   Thus $(\mathbf{E}[\,c\,], \sigma) \not\longrightarrow_{\mathsf{t}}$ **abort**. Then we can get the conclusion.
2. $\widetilde{C}$ is $\mathbf{E}[\,\mathbf{return}\ E\,]$.
   Thus $[\![E]\!]_\sigma = n$ and
   $$\begin{array}{l}\forall U.\ (U, \_) \in \Delta_1 \ \Rightarrow\ U(\mathsf{t}) = (\mathbf{end}, n)\,,\\ \forall U.\ (U, \_) \in \Delta_2 \ \Rightarrow\ U(\mathsf{t}) = (\mathbf{end}, n)\,.\end{array}$$
   Thus $\forall U.\ (U, \_) \in \Delta \ \Rightarrow\ U(\mathsf{t}) = (\mathbf{end}, n)$ and we are done.
3. $\widetilde{C}$ is $\mathbf{E}[\,\mathbf{linself}\,]$.
   Thus $\exists \Delta'_1.\ (\Delta_1 \rightarrow_{\mathsf{t}} \Delta'_1)$ and $\exists \Delta'_2.\ (\Delta_2 \rightarrow_{\mathsf{t}} \Delta'_2)$.
   Since $\Delta = \Delta_1 \cup \Delta_2$, we know $\Delta \rightarrow_{\mathsf{t}} (\Delta'_1 \cup \Delta'_2)$. Then we can get the conclusion.
4. $\widetilde{C}$ is $\mathbf{E}[\,\mathbf{commit}(p)\,]$.
   Thus $\mathsf{SpecExact}(p)$ and there exist Δ'_1 and Δ'_2 such that $(\sigma, \Delta_1)|_p = (\_, \Delta'_1)$ and $(\sigma, \Delta_2)|_p = (\_, \Delta'_2)$.
   By Lemma 61, we know $(\sigma, \Delta)|_p = (\_, \Delta'_1 \cup \Delta'_2)$. Then we can get the conclusion.
5. Other cases are similar.

So we get the conclusion. $\square$

Lemma 64 (Spec-Conjunction-Sound).
If $\models_{\mathsf{t}} \{p\}\widetilde{C}\{q\}$ and $\models_{\mathsf{t}} \{p'\}\widetilde{C}\{q'\}$, then $\models_{\mathsf{t}} \{p \oplus p'\}\widetilde{C}\{q \oplus q'\}$.

Proof: For any σ and Δ such that $(\sigma, \Delta) \models (p \oplus p')$, we know there exist Δ_1 and Δ_2 such that

$$(\sigma, \Delta_1) \models p,\ \ (\sigma, \Delta_2) \models p',\ \text{ and }\ \ \Delta = \Delta_1 \cup \Delta_2\,.$$

By the premises, we know

1. for any σ'_1 and Δ'_1, if $(\widetilde{C}, (\sigma, \Delta_1)) \hookrightarrow^*_{\mathsf{t}} (\mathbf{skip}, (\sigma'_1, \Delta'_1))$, then $(\sigma'_1, \Delta'_1) \models q$;
2. $(\widetilde{C}, (\sigma, \Delta_1)) \not\hookrightarrow^*_{\mathsf{t}}$ **abort**;
3. for any σ'_2 and Δ'_2, if $(\widetilde{C}, (\sigma, \Delta_2)) \hookrightarrow^*_{\mathsf{t}} (\mathbf{skip}, (\sigma'_2, \Delta'_2))$, then $(\sigma'_2, \Delta'_2) \models q'$;
4. $(\widetilde{C}, (\sigma, \Delta_2)) \not\hookrightarrow^*_{\mathsf{t}}$ **abort**.

Thus, for any σ' and Δ', if $(\widetilde{C}, (\sigma, \Delta)) \hookrightarrow^*_{\mathsf{t}} (\mathbf{skip}, (\sigma', \Delta'))$, by Lemma 62 and the above 2 and 4, we know: there exist Δ'_1 and Δ'_2 such that

$$\Delta' = \Delta'_1 \cup \Delta'_2\,,$$

$$(\widetilde{C}, (\sigma, \Delta_1)) \hookrightarrow_{\mathsf{t}}^{*} (\mathbf{skip}, (\sigma', \Delta'_1))\,,$$
$$(\widetilde{C}, (\sigma, \Delta_2)) \hookrightarrow_{\mathsf{t}}^{*} (\mathbf{skip}, (\sigma', \Delta'_2))\,.$$

From the above 1 and 3, we know

$$(\sigma', \Delta'_1) \models q \quad \text{and} \quad (\sigma', \Delta'_2) \models q'\,.$$

Thus we have $(\sigma', \Delta') \models q \oplus q'$.
Finally, by Lemmas 62 and 63, we know $(\widetilde{C}, (\sigma, \Delta)) \not\hookrightarrow_{\mathsf{t}}^{*} \mathbf{abort}$. Thus by Definition 38, we get the conclusion. □

C.3 Proof of Theorem 10 (Logic Soundness *w.r.t.* Contextual Refinement and Linearizability)

Theorem 10 is obtained immediately from Lemmas 41, 8 and 9.

D. Linking with Client Program Verification

As we mentioned in Sec. 4, our relational logic as an extension of LRG can be used to verify client code as well as object implementations. Moreover, since our logic ensures contextual refinement, it can provide us with "separation and information hiding" [25] over the object, but still keep enough information (*i.e.*, the abstract operations) about the method calls in concurrent client verification. To verify a program W, we could replace the object implementation with the abstract operations and verify the corresponding abstract program $\mathbb{W}$ instead. Below we will show a LINK rule which links object verification with client verification.

D.1 The Assertion Language for Client Verification

We first define the assertion language used to verify client code $\mathbb{W}$ after replacing concrete object implementation with abstract operations. We use different syntax to distinguish the assertions for client states and those for object states. The syntax of the assertions is given below.

$$(\mathit{CAssn})\ \ \mathbb{P}, \mathbb{Q}, \mathbb{I} ::= p \mid \boxed{p} \mid \mathbb{P} \wedge \mathbb{Q} \mid \mathbb{P} * \mathbb{Q} \mid \mathbb{P} \Rightarrow \mathbb{Q} \mid \ldots$$
$$(\mathit{CAct})\ \ \mathbb{R}, \mathbb{G} ::= R \mid \boxed{R} \mid \mathbb{R} \wedge \mathbb{R} \mid \mathbb{R} * \mathbb{R} \mid \ldots$$

Here $p \in \mathit{RelAss}$ and $R \in \mathit{RelAct}$ are an assertion and an action in the assertion language for linearizability verification. The semantics is defined as follows, where we use $\models_{\mathsf{L}}$ to represent the semantics in linearizability verification (Figures 9 and 10).

$$(\sigma_c, \theta) \models p \ \text{ iff } \ (\sigma_c, \emptyset) \models_{\mathsf{L}} p \wedge (\theta = \emptyset)$$
$$(\sigma_c, \theta) \models \boxed{p} \ \text{ iff } \ (\emptyset, \{(\emptyset, \theta)\}) \models_{\mathsf{L}} p \wedge (\sigma_c = \emptyset)$$
$$(\sigma_c, \theta) \models \mathbb{P} * \mathbb{Q} \ \text{ iff } \ \exists \sigma'_c, \theta', \sigma''_c, \theta''.\, (\sigma_c = \sigma'_c \uplus \sigma''_c) \wedge (\theta = \theta' \uplus \theta'') \wedge (\sigma'_c, \theta') \models \mathbb{P} \wedge (\sigma''_c, \theta'') \models \mathbb{Q}$$

$$((\sigma_c, \theta), (\sigma'_c, \theta')) \models R \ \text{ iff } \ ((\sigma_c, \emptyset), (\sigma'_c, \emptyset)) \models_{\mathsf{L}} R \wedge (\theta = \theta' = \emptyset)$$
$$((\sigma_c, \theta), (\sigma'_c, \theta')) \models \boxed{R} \ \text{ iff } \ ((\emptyset, \{(\emptyset, \theta)\}), (\emptyset, \{(\emptyset, \theta')\})) \models_{\mathsf{L}} R \wedge (\sigma_c = \sigma'_c = \emptyset)$$

Similarly, we can also define the assertion language at the concrete level, whose syntax is almost the same as that at the abstract level with one more assertion:

$$\mathbb{P}, \mathbb{Q}, \mathbb{I} \ ::= \ \ldots \mid \boxed{\varphi^{-1}(p)}$$

The semantics of assertions is defined similarly. Below we only show the semantics of the newly added assertion.

$$(\sigma_c, \sigma_o) \models' \boxed{\varphi^{-1}(p)} \ \text{ iff } \ (\emptyset, \{(\emptyset, \varphi(\sigma_o))\}) \models_{\mathsf{L}} p \wedge (\sigma_c = \emptyset)$$

D.2 The LINK Rule

$$\frac{\Pi \preceq_{\varphi} \Gamma \qquad \vdash \{p * \boxed{r}\}\mathbf{with}\ \Gamma\ \mathbf{do}\ C_1 \parallel \ldots \parallel C_n\{q * \boxed{\mathbf{true}}\}}{\vdash \{p * \boxed{\varphi^{-1}(r)}\}\mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \parallel \ldots \parallel C_n\{q * \boxed{\mathbf{true}}\}}$$

The LINK rule simply says that if we know Π is linearizable *w.r.t.* Γ (*e.g.*, verified in our relational logic), then the proof of the partial correctness of the abstract client can be directly translated to the proof for the concrete client. It relates the verification of Π to client verification.

To prove the soundness of the LINK rule, we first define the judgment semantics as follows.

Definition 65 (Judgment Semantics). $\models \{\mathbb{P}\}\mathbb{W}\{\mathbb{Q}\}$ iff, for any σ_c, θ and $\mathbb{K}$, if $(\sigma_c, \theta) \models \mathbb{P}$ and $\forall \mathsf{t}.\ \mathbb{K}(\mathsf{t}) = \circ$, the following are true:

1. for any σ'_c and θ', if $(\mathbb{W}, (\sigma_c, \theta, \mathbb{K})) \xrightarrow{-}\!\!{}^{*} (\mathbf{skip}, (\sigma'_c, \theta', \_))$, then $(\sigma'_c, \theta') \models \mathbb{Q}$;
2. $(\mathbb{W}, (\sigma_c, \theta, \mathbb{K})) \not\longrightarrow^{*} \mathbf{abort}$.

$\models \{\mathbb{P}\}W\{\mathbb{Q}\}$ is defined similarly.

The following lemma says that from the contextual refinement $\Pi \sqsubseteq_{\varphi} \Gamma$, we know the termination of the concrete client implies the termination of the abstract client, and the final client states are the same.

Lemma 66. For any n, $C_1, \ldots, C_n$, σ_c, σ_o and θ, if

(1) $\Pi \sqsubseteq_{\varphi} \Gamma$,
(2) $(\mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \parallel \ldots \parallel C_n, (\sigma_c, \sigma_o, \mathcal{K})) \overset{-}{\longmapsto}{}^{*} (\mathbf{skip}, (\sigma''_c, \_, \_))$, where $\forall \mathsf{t}.\ \mathcal{K}(\mathsf{t}) = \circ$,

then $(\mathbf{with}\ \Gamma\ \mathbf{do}\ C_1 \parallel \ldots \parallel C_n, (\sigma_c, \theta, \mathbb{K})) \xrightarrow{-}{}^{*} (\mathbf{skip}, (\sigma''_c, \_, \_))$, where $\forall \mathsf{t}.\ \mathbb{K}(\mathsf{t}) = \circ$ and $\varphi(\sigma_o) = \theta$.

Proof: We assume there is an instruction **print_state** to print out the whole client state, which simply generates the observable event $(\mathsf{t}, \mathbf{out}, \sigma_c)$ at the current client state σ_c.

From $C_1, \ldots, C_n$, we construct $C'_1, \ldots, C'_n$, and from σ_c, we construct σ'_c and a function f such that $\sigma_c = f(\sigma'_c)$, and also the following hold:

(3) If $(\mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \parallel \ldots \parallel C_n, (\sigma_c, \sigma_o, \mathcal{K})) \overset{H}{\longmapsto}{}^{*} (\mathbf{skip}, (\sigma''_c, \_, \_))$, then there exist H' and σ'''_c such that $\sigma''_c = f(\sigma'''_c)$, $H' = H :: (\_, \mathbf{out}, \sigma'''_c)$ and $(\mathbf{let}\ \Pi\ \mathbf{in}\ C'_1 \parallel \ldots \parallel C'_n, (\sigma'_c, \sigma_o, \mathcal{K})) \overset{H'}{\longmapsto}{}^{*} (\mathbf{skip}, (\sigma'''_c, \_, \_))$.
(4) If $(\mathbf{with}\ \Gamma\ \mathbf{do}\ C'_1 \parallel \ldots \parallel C'_n, (\sigma'_c, \theta, \mathbb{K})) \xrightarrow{H_a}{}^{*} \_$ and $\mathsf{last}(\mathsf{get\_obsv}(H_a)) = (\_, \mathbf{out}, \sigma'''_c)$, then $(\mathbf{with}\ \Gamma\ \mathbf{do}\ C_1 \parallel \ldots \parallel C_n, (\sigma_c, \theta, \mathbb{K})) \xrightarrow{-}{}^{*} (\mathbf{skip}, (f(\sigma'''_c), \_, \_))$.

Then, we can prove the lemma as follows:
From (2) and (3), we know there exist H' and σ'''_c such that $\sigma''_c = f(\sigma'''_c)$, $\mathsf{last}(\mathsf{get\_obsv}(H')) = (\_, \mathbf{out}, \sigma'''_c)$ and $(\mathbf{let}\ \Pi\ \mathbf{in}\ C'_1 \parallel \ldots \parallel C'_n, (\sigma'_c, \sigma_o, \mathcal{K})) \overset{H'}{\longmapsto}{}^{*} (\mathbf{skip}, (\sigma'''_c, \_, \_))$.
From (1), we know there exists H_a such that $\mathsf{last}(\mathsf{get\_obsv}(H_a)) = (\_, \mathbf{out}, \sigma'''_c)$ and $(\mathbf{with}\ \Gamma\ \mathbf{do}\ C'_1 \parallel \ldots \parallel C'_n, (\sigma'_c, \theta, \mathbb{K})) \xrightarrow{H_a}{}^{*} \_$, where $\forall \mathsf{t}.\ \mathbb{K}(\mathsf{t}) = \circ$.
From (4), we get the conclusion.

We construct $C'_1, \ldots, C'_n$, σ'_c and the function f as follows:

- If $n = 1$, let $C'_1 = (C_1; \mathbf{print\_state})$ and $\sigma'_c = \sigma_c$. The function f is an identity function.
- If $n \geq 2$, we pick $n - 1$ fresh variables $\mathtt{d}_2, \ldots, \mathtt{d}_n$, and let
$$C'_1 = (C_1; \mathbf{if}\ (\mathtt{d}_2 \&\& \ldots \&\& \mathtt{d}_n)\ \mathbf{print\_state}),$$
and for any $i \in [2..n]$, let $C'_i = (C_i; \mathtt{d}_i := \mathbf{true})$. Let
$$\sigma'_c = \sigma_c \uplus \{\mathtt{d}_2 \rightsquigarrow \mathbf{false}, \ldots, \mathtt{d}_n \rightsquigarrow \mathbf{false}\}.$$
The function f is a projection which removes $\mathtt{d}_2, \ldots \mathtt{d}_n$.

$$\frac{\Pi \preceq_{\varphi} \Gamma \quad \vdash \{p * \boxed{r}\}\textbf{with } \Gamma \textbf{ do } C_1 \,\|\, \ldots \,\|\, C_n\{q * \boxed{\textbf{true}}\}}{\vdash \{p * \boxed{\varphi^{-1}(r)}\}\textbf{let } \Pi \textbf{ in } C_1 \,\|\, \ldots \,\|\, C_n\{q * \boxed{\textbf{true}}\}} \ (\text{LINK}) \qquad \frac{p \Rightarrow (E = E) * \textbf{true} \wedge (E' = X) * \textbf{true} \quad (E, r)\llbracket\Gamma(f)\rrbracket^{p}(E', r')}{\vdash_{\mathsf{t},\Gamma} \{p * (x = \_) * \boxed{r}\}x := f(E)\{p * (x = X) * \boxed{r'}\}} \ (\text{CALL})$$

$$\frac{\begin{array}{c}\vdash_{\mathsf{t},\Gamma} \{p * \boxed{r}\}x := f(E)\{q * \boxed{r'}\} \qquad p * \boxed{r} \vee q * \boxed{r'} \Rightarrow \mathbb{I} * \textbf{true} \\ [p] * \boxed{r \ltimes r'} \Rightarrow \mathbb{G} * \mathsf{True} \qquad (p \ltimes q) * \boxed{[r']} \Rightarrow \mathbb{G} * \mathsf{True} \qquad \mathbb{I} \rhd \mathbb{G}\end{array}}{[\mathbb{I}], \mathbb{G}, \mathbb{I} \vdash_{\mathsf{t},\Gamma} \{p * \boxed{r}\}x := f(E)\{q * \boxed{r'}\}} \ (\text{CALL-G}) \qquad \frac{\begin{array}{c}[\mathbb{I}], \mathbb{G}, \mathbb{I} \vdash_{\mathsf{t},\Gamma} \{\mathbb{P}\}x := f(E)\{\mathbb{Q}\} \\ \mathsf{Sta}(\{\mathbb{P}, \mathbb{Q}\}, \mathbb{R} * \mathsf{Id}) \qquad \mathbb{I} \rhd \mathbb{R}\end{array}}{\mathbb{R}, \mathbb{G}, \mathbb{I} \vdash_{\mathsf{t},\Gamma} \{\mathbb{P}\}x := f(E)\{\mathbb{Q}\}} \ (\text{CALL-R})$$

$$\frac{\forall i \in [1..n] \quad \mathbb{R}_i, \mathbb{G}_i, \mathbb{I} \vdash_{i,\Gamma} \{\mathbb{P}_i * \mathbb{P}\}C_i\{\mathbb{Q}_i * \mathbb{Q}'_i\} \quad \mathbb{R}_i = \bigvee_{j \neq i} \mathbb{G}_j \quad \mathbb{I} \rhd \mathbb{R}_i \quad \mathbb{P} \vee \mathbb{Q}'_i \Rightarrow \mathbb{I}}{\vdash \{\mathbb{P}_1 * \ldots * \mathbb{P}_n * \mathbb{P}\}\textbf{with } \Gamma \textbf{ do } C_1 \,\|\, \ldots \,\|\, C_n\{\mathbb{Q}_1 * \ldots * \mathbb{Q}_n * (\mathbb{Q}'_1 \wedge \ldots \wedge \mathbb{Q}'_n)\}} \ (\text{PAR})$$

Auxiliary definition:

$$\begin{array}{l}(E, r)\llbracket\Gamma(f)\rrbracket^{p}(E', r') \ \text{ iff} \\ \forall \sigma, \theta, n, \theta', n'.\, (\sigma \models p) \wedge (\theta \models r) \wedge (\llbracket E\rrbracket_{\sigma} = n) \wedge (\gamma(n)(\theta) = (n', \theta')) \\ \Longrightarrow \ (\theta' \models r') \wedge (\llbracket E'\rrbracket_{\sigma} = n')\end{array}$$

Figure 22. LRG-Style Inference Rules for Client Verification

For both cases, we can prove (3) and (4) hold.
Then we are done. □

Lemma 67 (Soundness of LINK Rule). Let $W = \textbf{let } \Pi \textbf{ in } C_1 \,\|\, \ldots \,\|\, C_n$ and $\mathbb{W} = \textbf{with } \Gamma \textbf{ do } C_1 \,\|\, \ldots \,\|\, C_n$. If $\Pi \preceq_{\varphi} \Gamma$, and $\models \{p * \boxed{r}\}\mathbb{W}\{q * \boxed{\textbf{true}}\}$, then $\models \{p * \boxed{\varphi^{-1}(r)}\}W\{q * \boxed{\textbf{true}}\}$.

Proof: For any σ_c, σ_o and $\mathcal{K}$, if $(\sigma_c, \sigma_o) \models p * \boxed{\varphi^{-1}(r)}$ and $\forall \mathsf{t}.\, \mathcal{K}(\mathsf{t}) = \circ$, we prove the following:

(1) for any σ'_c and σ'_o, if $(W, (\sigma_c, \sigma_o, \mathcal{K})) \overset{-}{\longmapsto}{}^{*} (\textbf{skip}, (\sigma'_c, \sigma'_o, \_))$, then $(\sigma'_c, \emptyset) \models_{\mathsf{L}} q$;
(2) $(W, (\sigma_c, \sigma_o, \mathcal{K})) \not\longmapsto^{*} \textbf{abort}$.

Since $(\sigma_c, \theta) \models p * \boxed{\varphi^{-1}(r)}$, we know $(\sigma_c, \emptyset) \models_{\mathsf{L}} p$ and there exists θ such that $\varphi(\sigma_o) = \theta$ and $(\emptyset, \{(\emptyset, \theta)\}) \models_{\mathsf{L}} r$.
Since $\Pi \preceq_{\varphi} \Gamma$, we know $\Pi \sqsubseteq_{\varphi} \Gamma$, thus $\mathcal{O}\llbracket W, (\sigma_c, \sigma_o)\rrbracket \subseteq \mathcal{O}\llbracket \mathbb{W}, (\sigma_c, \theta)\rrbracket$.
For (1), by the above Lemma 66, we know there exists θ' such that

$$(\mathbb{W}, (\sigma_c, \theta, \mathbb{K})) \overset{-}{\Longrightarrow}{}^{*} (\textbf{skip}, (\sigma'_c, \theta', \_)),$$

where $\forall \mathsf{t}.\, \mathbb{K}(\mathsf{t}) = \circ$.
Since $\models \{p * \boxed{r}\}\mathbb{W}\{q * \boxed{\textbf{true}}\}$, we know $(\sigma'_c, \emptyset) \models_{\mathsf{L}} q$.
For (2), since $\models \{p * \boxed{r}\}\mathbb{W}\{q * \boxed{\textbf{true}}\}$, we know

$$(\mathbb{W}, (\sigma_c, \theta, \mathbb{K})) \not\Longrightarrow^{*} \textbf{abort}.$$

Since $\mathcal{O}\llbracket W, (\sigma_c, \sigma_o)\rrbracket \subseteq \mathcal{O}\llbracket \mathbb{W}, (\sigma_c, \theta)\rrbracket$, we get the conclusion. □

D.3 Client Verification

We show the inference rules for $\vdash \{\mathbb{P}\}\mathbb{W}\{\mathbb{Q}\}$ in Figure 22, including the rules for method calls and parallel compositions. The rules allow us the reason about the client code as if it was using the abstract object. The current inference rules in Figure 22 are based on the plain LRG [8], but they can also be adapted from the standard rely-guarantee-style rules [17] or CSL-style rules [23].

E. More Examples

In Section 6, we have sketched the proofs of three examples: the pair snapshot, MS lock-free queue and the CCAS algorithms. In this section, we give the proofs of the other nine examples we have verified, and the complete proofs of the MS lock-free queue and the CCAS algorithm.

To make the proofs more compact and readable, we allow variables to occur in separate assertions which are starring together, for example, the following notation is allowed:

$$(\mathtt{a} \mapsto \mathtt{b}) * (\mathtt{b} \mapsto \mathtt{null}),$$

which can be viewed as a shorthand for (where l is an integer)

$$\exists l.\, (\mathtt{a} \mapsto l) * (l \mapsto \mathtt{null}) * (\mathtt{b} = l).$$

Besides, when local variables are unused anymore, we can simply omit them in assertions. In other words, all our assertions are implicitly starring the ownership of unused local variables, including the formal arguments for methods.

E.1 Treiber Stack

We have introduced Treiber stack in Section 2.1. Here we give its complete implementation in Figure 23(a). The algorithm does not use fancy techniques such as helping mechanism and future-dependent linearization points. The abstract PUSH and POP operations defined in Figure 23(b) manipulate the abstract mathematical list Stk, and when popping from an empty stack, the POP returns EMPTY.

We define the precise invariant, the rely and the guarantee in Figure 24, and show the proof in Figure 25, where we highlight the instrumented auxiliary commands.

The invariant I in Figure 24 maps the value sequence A of the concrete list pointed to by S (denoted by $\mathsf{ls}(\mathtt{S}, A, \mathtt{null})$) to the abstract stack Stk. To ensure there is no "ABA" problem [15], we follow Turon and Wand [29] and introduce a write-only auxiliary variable GN to remember the nodes which used to be on the stack but no longer are. The precise invariant for shared states should include those garbage nodes (garb). GN does not affect the behaviors of the implementation and is introduced for verification only.

The guarantee includes the push and the pop actions. At the concrete side, the actions correspond to the linearization points: line 6 for push and line 17 for pop in Figure 23(a). Note that when popping a node, we also add the node to GN. The rely of a thread is the same as its guarantee.

The proof in Figure 25 is straightforward. We let the abstract operations be executed simultaneously with the concrete code at linearization points, so that we can ensure when the concrete code returns, we have the matched abstract return values. Note that when

$I \overset{\text{def}}{=} \exists A.\, \mathsf{ls}(\mathtt{S}, A, \mathtt{null}) * (\mathtt{Stk} \Rrightarrow A) * \mathsf{garb}$

$\mathsf{node}(x, v, y) \overset{\text{def}}{=} x \mapsto (v, y) \qquad \mathsf{node}(x) \overset{\text{def}}{=} \mathsf{node}(x, \_, \_) \qquad \mathsf{garb} \overset{\text{def}}{=} \circledast_{x \in \mathtt{GN}}.\mathsf{node}(x)$

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

$R = G \overset{\text{def}}{=} [\mathsf{Push} \vee \mathsf{Pop}]_I$

$\mathsf{Push} \overset{\text{def}}{=} \exists x, y, v, A.\, ((\mathtt{S} = y) * \mathtt{Stk} \Rrightarrow A) \ltimes ((\mathtt{S} = x) * \mathsf{node}(x, v, y) * \mathtt{Stk} \Rrightarrow v :: A)$

$\mathsf{Pop} \overset{\text{def}}{=} \exists x, y, v, A, S_g.\, ((\mathtt{S} = x) * \mathsf{node}(x, v, y) * (\mathtt{GN} = S_g) * \mathtt{Stk} \Rrightarrow v :: A) \ltimes ((\mathtt{S} = y) * \mathsf{node}(x, v, y) * (\mathtt{GN} = S_g \cup \{x\}) * \mathtt{Stk} \Rrightarrow A)$

Figure 24. Precise Invariant, Rely and Guarantee of Treiber Stack

```
struct Node {
  int data;
  struct Node *next;
}
struct Stack {
  struct Node *S;
}

void push(int v) :
     local d, x, t;
  1  x := cons(v, null);
  2  d := 0;
  3  while (d = 0) {
  4    t := S;
  5    x.next := t;
  6    d := cas(&S, t, x);
  7  }

int pop() :
     local v, d, x, t;
  8  d := 0;
  9  while (d = 0) {
 10    t := S;
 11    if (t = null) {
 12      v := EMPTY;
 13      d := 1;
 14    } else {
 15      v := t.data;
 16      x := t.next;
 17      d := cas(&S,t,x);
 18    }
 19  }
 20  return v;
```

(a) Concrete Implementation

$\theta \in \{\mathtt{Stk}\} \to \mathit{List}(\mathit{Int})$

$\mathtt{PUSH}(v)(\theta) \overset{\text{def}}{=} (\mathbf{void}, \theta\{\mathtt{Stk} \rightsquigarrow v :: \theta(\mathtt{Stk})\})$

$$\mathtt{POP}(\_)(\theta) \overset{\text{def}}{=} \begin{cases} (v, \theta\{\mathtt{Stk} \rightsquigarrow S\}) & \text{if } \theta(\mathtt{Stk}) = v :: S \\ \mathtt{EMPTY} & \text{otherwise} \end{cases}$$

(b) Abstract Operations

Figure 23. Treiber Stack Code

```
push(v):
     local d, x, t;
    $\{ I * \mathtt{cid} \rightarrowtail (\mathrm{PUSH}, \mathrm{v}) \}$
 1   x := cons(v, null);
    $\{ I * \mathsf{node}(\mathrm{x}, \mathrm{v}, \mathtt{null}) * \mathtt{cid} \rightarrowtail (\mathrm{PUSH}, \mathrm{v}) \}$
 2   d := 0;
    $\left\{ \begin{array}{l} ((\mathrm{d} = 0) * I * \mathsf{node}(\mathrm{x}, \mathrm{v}, \mathtt{null}) * \mathtt{cid} \rightarrowtail (\mathrm{PUSH}, \mathrm{v})) \\ \quad \vee ((\mathrm{d} = 1) * I * \mathtt{cid} \rightarrowtail (\mathbf{end}, \mathbf{void})) \end{array} \right\}$
 3   while (d = 0) {
      $\{ (\mathrm{d} = 0) * I * \mathsf{node}(\mathrm{x}, \mathrm{v}, \mathtt{null}) * \mathtt{cid} \rightarrowtail (\mathrm{PUSH}, \mathrm{v}) \}$
 4     t := S;
 5     x.next := t;
      $\{ (\mathrm{d} = 0) * I * \mathsf{node}(\mathrm{x}, \mathrm{v}, \mathrm{t}) * \mathtt{cid} \rightarrowtail (\mathrm{PUSH}, \mathrm{v}) \}$
 6     < d := cas(&S, t, x);  if (d) linself; >
 7   }
    $\{ (\mathrm{d} = 1) * I * \mathtt{cid} \rightarrowtail (\mathbf{end}, \mathbf{void}) \}$

IntSet GN;
//Auxiliary global variable for verification: popped garbage nodes

pop():
     local v, d, x, t;
    $\{ I * \mathtt{cid} \rightarrowtail \mathrm{POP} \}$
 8   d := 0;
    $\left\{ \begin{array}{l} ((\mathrm{d} = 0) * I * \mathtt{cid} \rightarrowtail \mathrm{POP}) \\ \quad \vee ((\mathrm{d} = 1) * I * \mathtt{cid} \rightarrowtail (\mathbf{end}, \mathrm{v})) \end{array} \right\}$
 9   while (d = 0) {
      $\{ (\mathrm{d} = 0) * I * \mathtt{cid} \rightarrowtail \mathrm{POP} \}$
10     < t := S;  if (t = null) linself; >
      $\left\{ \begin{array}{l} ((\mathrm{d} = 0) * I * (\mathrm{t} = \mathtt{null}) * \mathtt{cid} \rightarrowtail (\mathbf{end}, \mathrm{EMPTY})) \\ \quad \vee ((\mathrm{d} = 0) * (I \wedge \mathsf{node}(\mathrm{t}) * \mathsf{true}) * \mathtt{cid} \rightarrowtail \mathrm{POP}) \end{array} \right\}$
11     if (t = null) {
        $\{ (\mathrm{d} = 0) * I * (\mathrm{t} = \mathtt{null}) * \mathtt{cid} \rightarrowtail (\mathbf{end}, \mathrm{EMPTY}) \}$
12       v := EMPTY;
13       d := 1;
        $\{ (\mathrm{d} = 1) * I * \mathtt{cid} \rightarrowtail (\mathbf{end}, \mathrm{v}) \}$
14     } else {
        $\{ (\mathrm{d} = 0) * (I \wedge \mathsf{node}(\mathrm{t}) * \mathsf{true}) * \mathtt{cid} \rightarrowtail \mathrm{POP} \}$
15       v := t.data;
16       x := t.next;
        $\{ (\mathrm{d} = 0) * (I \wedge \mathsf{node}(\mathrm{t}, \mathrm{v}, \mathrm{x}) * \mathsf{true}) * \mathtt{cid} \rightarrowtail \mathrm{POP} \}$
17       < d := cas(&S,t,x);  GN := GN ∪ {t};
18         if (d) linself; >
        $\left\{ \begin{array}{l} ((\mathrm{d} = 0) * I * \mathtt{cid} \rightarrowtail \mathrm{POP}) \\ \quad \vee ((\mathrm{d} = 1) * I * \mathtt{cid} \rightarrowtail (\mathbf{end}, \mathrm{v})) \end{array} \right\}$
19     }
20   }
    $\{ (\mathrm{d} = 1) * I * \mathtt{cid} \rightarrowtail (\mathbf{end}, \mathrm{v}) \}$
21   return v;
```

Figure 25. Proof Outline for Treiber Stack for Thread `cid`

popping from an empty stack, the linearization point is at line 10, where the thread reads the stack pointer. Although at line 10 the stack is empty, the thread would realize that the pointer is `null` at a later time (line 11 succeeds). We cannot linearize the operation at line 11 or later, because at that time the stack may be not empty anymore: the environment may have done pushes during the time.

E.2 HSY Elimination-Based Stack

As explained in Sec. 2.2, HSY stack uses elimination, allowing a push and a pop operations to help each other. We show the complete implementation in Figure 26. The abstract `PUSH` and `POP` operations are totally the same as those for Treiber stack.

To verify HSY stack in our logic, we first define the precise invariant and the rely/guarantee conditions over the shared relational states in Fig. 27. The invariant I contains three parts. As in Treiber stack, the stack part stkInv maps the value sequence A of the concrete list pointed to by `S` to the abstract stack `Stk`. We also introduce a write-only auxiliary variable `GN` to remember the nodes which were popped from the stack. The precise invariant for shared states should include those garbage nodes (garb). The last part $\mathsf{collInv}$ specifies the collision array `coll` used by the algorithm to choose a random slot in the `loc` array for elimination. It is actually used as an optimization of `him:=rand()` at line 7 in Fig. 1(b), and does not affect our main proofs.

The most important part in I is the elimination part locInv. It describes the global `loc` array and the threads whose descriptors are in the array to be helped by others (Locs). It also contains a set of thread descriptors (Ds). Once allocated, a thread descriptor will never be reclaimed in the algorithm (even when it is out of date and unreachable from `loc` anymore). We introduce an auxiliary variable `D` to remember all of them. Here we use $\mathsf{d}(p, id, op, n)$ to mean p points to the descriptor (id, op, n). Moreover, locInv contains a set of pushing threads which have been eliminated by others ($\mathsf{EPushes}$). For these threads, after they put their descriptors in the `loc` array, some popping threads come and clear their slots to inform them that they have been eliminated. These threads need to check their slots to get this information. After the check, they can get back their abstract operations (now they must be $(\mathbf{end}, \mathbf{void})$) and return. We use the auxiliary variable `EPush` to collect these pushing threads who have not returned their eliminated operations. Note the three auxiliary variables `GN`, `D` and `EPush` are all write-only and introduced for verification only.

The rely of thread $\mathtt{t}$ includes the guarantees of all the other threads: $R_{\mathtt{t}} \overset{\text{def}}{=} \bigvee_{\mathtt{t}' \neq \mathtt{t}} G_{\mathtt{t}'}$. The guarantee of thread $\mathtt{t}$, shown in Fig. 27, can be divided into actions on the central stack (fenced by stkInv), on the `loc` array (fenced by locInv) and on the `coll` array (fenced by $\mathsf{collInv}$). As usual, the thread can do Push and Pop on the central stack. It can also update the `coll` array ($\mathsf{UpdColl}$). For the locInv part, the thread can allocate its descriptor (AllocD), place its descriptor in the `loc` array (PlaceD), and remove its descriptor (RmvD). It can also eliminate a descriptor in `loc` array ($\mathsf{ElimPush}$ if it itself is a pop, or $\mathsf{ElimPop}$ otherwise). If it is eliminated by another thread, it can finish the operation by setting the return value for pop ($\mathsf{SetPopV}$) and removing the descriptor in its slot, or simply returning its result for push ($\mathsf{FinishPush}$). Since we treat abstract operations as auxiliary states, we can have *ownership transfers* on them, *e.g.*, $\mathsf{PlaceD}_{\mathtt{t}}$ makes $\mathtt{t}$'s abstract operation become shared, while $\mathsf{RmvD}_{\mathtt{t}}$ transfers it back to the thread local state.

We give the proof outlines of the implementation in Figure 30 (for the top-level code `StackOp`), Figure 28 (for the elimination part `TryCollision`), and Figure 29 (for `FinishCollision`). The proof is straightforward, and we only explain the most important part for elimination below.

Fig. 28 shows the proof of the core code for elimination, `TryCollision`, which includes line 10 of Fig. 1(b) for a push

```
struct Node {
  int data;
  struct Node *next;
}
struct Stack {
  struct Node *S;
}
struct ThrdInfo {
  int id;
  int op;
  int data;
}
ThrdInfo *loc[1..thrdNum];
int coll[1..size];

void push(int v) :
      local p;
  1   p := cons(cid, PUSH, v);
  2   StackOp(p);

int pop() :
      local p;
  3   p := cons(cid, POP, 0);
  4   StackOp(p);
  5   return p.data;

void StackOp(ThrdInfo p) :
      local him, q, pos;
  6   while (true) {
  7     if (TryStackOp(p))
  8       return;
  9     loc[cid] := p;
 10     pos := GetPosition(p);
 11     him := coll[pos];
 12     while (!cas(&coll[pos], him, cid))
 13       him := coll[pos];
 14     if (1 <= him <= thrdNum) {
 15       q := loc[him];
 16       if (q != null && q.id = him && q.op != p.op)
 17         if (cas(&loc[cid], p, null)) {
 18           if (TryCollision(p, q))
 19             return;
 20           else
 21             continue;
 22         } else {
 23           FinishCollision(p);
 24           return; }
 25     }
 26     if (!cas(&loc[cid], p, null)) {
 27       FinishCollision(p); return;
 28     }
 29   }

bool TryCollision(ThrdInfo p, q) :
      local b;
 30   if (p.op = PUSH) {
 31     b := cas(&loc[q.id], q, p);
 32   } else if (p.op = POP) {
 33     b := cas(&loc[q.id], q, null);
 34     if (b) p.data := q.data;
 35   }
 36   return b;

void FinishCollision(ThrdInfo p) :
 37   if (p.op = POP) {
 38     p.data := loc[cid].data;
 39     loc[cid] := null;
 40   }
```

Figure 26. HSY Stack Code

$$
\begin{array}{l}
I \overset{\text{def}}{=} \mathsf{stkInv} * \mathsf{locInv} * \mathsf{collInv} \\
\mathsf{stkInv} \overset{\text{def}}{=} \exists A.\, \mathsf{ls}(\mathtt{S}, A, \mathtt{null}) * (\mathtt{Stk} \Mapsto A) * \mathsf{garb} \qquad \mathsf{garb} \overset{\text{def}}{=} (\circledast_{x\in \mathtt{GN}}.\mathsf{node}(x)) \\
\mathsf{locInv} \overset{\text{def}}{=} \mathsf{Locs} * \mathsf{Ds} * \mathsf{EPushes} \qquad \mathsf{Locs} \overset{\text{def}}{=} \circledast_{t\in[1..\mathtt{thrdNum}]}.((\mathtt{loc}[t] = \mathtt{null}) \vee ((\mathtt{loc}[t] \neq \mathtt{null}) * (t \rightarrowtail \_))) \\
\mathsf{d}(p, id, op, n) \overset{\text{def}}{=} p \mapsto (id, op, n) \wedge id \in \mathit{ThrdID} \wedge op \in \{\mathtt{PUSH}, \mathtt{POP}\} \qquad \mathsf{Ds} \overset{\text{def}}{=} \circledast_{p\in\mathtt{D}}.\mathsf{d}(p, \_, \_, \_) \\
\mathsf{EPushes} \overset{\text{def}}{=} \circledast_{t\in\mathtt{EPush}}.(t \rightarrowtail (\mathbf{end}, \mathbf{void})) \qquad \mathsf{collInv} \overset{\text{def}}{=} \circledast_{i\in[1..\mathtt{size}]}.(\mathtt{coll}[i] = \_)
\end{array}
$$

$$
\begin{array}{l}
R_{\mathsf{t}} \overset{\text{def}}{=} \bigvee_{\mathsf{t}'\neq\mathsf{t}} G_{\mathsf{t}'} \\
G_{\mathsf{t}} \overset{\text{def}}{=} [\mathsf{Push} \vee \mathsf{Pop}]_{\mathsf{stkInv}} * [\mathsf{AllocD}_{\mathsf{t}} \vee \mathsf{PlaceD}_{\mathsf{t}} \vee \mathsf{RmvD}_{\mathsf{t}} \vee \mathsf{ElimPush}_{\mathsf{t}} \vee \mathsf{ElimPop}_{\mathsf{t}} \vee \mathsf{SetPopV}_{\mathsf{t}} \vee \mathsf{FinishPush}_{\mathsf{t}}]_{\mathsf{locInv}} * [\mathsf{UpdColl}]_{\mathsf{collInv}} \\
\mathsf{Push} \overset{\text{def}}{=} \exists x, y, v, A.\, ((\mathtt{S} = y) * \mathtt{S} \Mapsto A) \ltimes ((\mathtt{S} = x) * \mathsf{node}(x, v, y) * \mathtt{S} \Mapsto v :: A) \\
\mathsf{Pop} \overset{\text{def}}{=} \exists x, y, v, A, S_g.\, ((\mathtt{S} = x) * \mathsf{node}(x, v, y) * (\mathtt{GN} = S_g) * \mathtt{S} \Mapsto v :: A) \ltimes ((\mathtt{S} = y) * \mathsf{node}(x, v, y) * (\mathtt{GN} = S_g \cup \{x\}) * \mathtt{S} \Mapsto A) \\
\mathsf{AllocD}_{\mathsf{t}} \overset{\text{def}}{=} \exists p, S_d.\, (\mathsf{emp} * (\mathtt{D} = S_d)) \ltimes (\mathsf{d}(p, \mathsf{t}, \_, \_) * (\mathtt{D} = S_d \cup \{p\})) \\
\mathsf{PlaceD}_{\mathsf{t}} \overset{\text{def}}{=} \exists p, op, n.\, ((\mathtt{loc}[\mathsf{t}] = \mathtt{null}) * \mathsf{d}(p, \mathsf{t}, op, n) \wedge p \in \mathtt{D}) \ltimes \mathsf{notDone}(p, \mathsf{t}, op, n) \\
\mathsf{RmvD}_{\mathsf{t}} \overset{\text{def}}{=} ((\mathtt{loc}[\mathsf{t}] \neq \mathtt{null}) * (\mathsf{t} \rightarrowtail \_)) \ltimes (\mathtt{loc}[\mathsf{t}] = \mathtt{null}) \\
\mathsf{ElimPush}_{\mathsf{t}} \overset{\text{def}}{=} \exists i, q, n, S_e.\, (\mathsf{notDone}(q, i, \mathtt{PUSH}, n) * (\mathtt{EPush} = S_e) \wedge (i \neq \mathsf{t})) \ltimes (\mathsf{elimPush}(q, i, n) * (\mathtt{EPush} = S_e \cup \{i\})) \\
\mathsf{ElimPop}_{\mathsf{t}} \overset{\text{def}}{=} \exists i, p, q, n, n'.\, (\mathsf{notDone}(q, i, \mathtt{POP}, n) * \mathsf{d}(p, \mathsf{t}, \mathtt{PUSH}, n') \wedge (i \neq \mathsf{t}) \wedge (p \in \mathtt{D})) \ltimes \mathsf{elimPop}(q, i, n, p, \mathsf{t}, n') \\
\mathsf{SetPopV}_{\mathsf{t}} \overset{\text{def}}{=} \exists p.\, (\mathsf{d}(p, \mathsf{t}, \mathtt{POP}, \_) \wedge (p \in \mathtt{D})) \ltimes \mathsf{d}(p, \mathsf{t}, \mathtt{POP}, \_) \\
\mathsf{FinishPush}_{\mathsf{t}} \overset{\text{def}}{=} \exists S.\, ((\mathtt{EPush} = S \cup \{\mathsf{t}\}) * \mathsf{t} \rightarrowtail (\mathbf{end}, \mathbf{void})) \ltimes (\mathtt{EPush} = S) \\
\mathsf{UpdColl} \overset{\text{def}}{=} \exists i.\, (\mathtt{coll}[i] = \_) \ltimes (\mathtt{coll}[i] = \_)
\end{array}
$$

$$
\begin{array}{l}
\mathsf{notDone}(p, t, op, n) \overset{\text{def}}{=} \mathsf{d}(p, t, op, n) * (\mathtt{loc}[t] = p) * (t \rightarrowtail (op, n)) \wedge (p \in \mathtt{D}) \\
\mathsf{elimPush}(p, t, n) \overset{\text{def}}{=} \mathsf{d}(p, t, \mathtt{PUSH}, n) * (\mathtt{loc}[t] = \mathtt{null}) * (t \rightarrowtail (\mathbf{end}, \mathbf{void})) \wedge (p \in \mathtt{D}) \\
\mathsf{elimPop}(p, t, n, q, t', n') \overset{\text{def}}{=} \mathsf{d}(p, t, \mathtt{POP}, n) * (\mathtt{loc}[t] = q) * \mathsf{d}(q, t', \mathtt{PUSH}, n') * (t \rightarrowtail (\mathbf{end}, n')) \wedge (t \neq t') \wedge (p, q \in \mathtt{D}) \\
I' \overset{\text{def}}{=} (\mathsf{locInv} \wedge \mathsf{locSubsetD}) * \mathsf{collInv} \qquad \mathsf{locSubsetD} \overset{\text{def}}{=} \forall t, p.\, ((\mathtt{loc}[t] = p) \wedge (p \neq \mathtt{null})) \Rightarrow p \in \mathtt{D} \\
\mathsf{notInLoc}(p, op, n) \overset{\text{def}}{=} I' \wedge (\mathtt{loc}[\mathtt{cid}] = \mathtt{null}) * \mathsf{d}(p, \mathtt{cid}, op, n) * \mathsf{true} \wedge (p \in \mathtt{D}) \\
\mathsf{begin}(p, op) \overset{\text{def}}{=} \exists n.\, \mathsf{notInLoc}(p, op, n) * (\mathtt{cid} \rightarrowtail (op, n)) \\
\mathsf{endPush}(p) \overset{\text{def}}{=} \mathsf{notInLoc}(p, \mathtt{PUSH}, \_) * (\mathtt{cid} \rightarrowtail (\mathbf{end}, \mathbf{void})) \qquad \mathsf{endPop}(p) \overset{\text{def}}{=} \exists n.\, \mathsf{notInLoc}(p, \mathtt{POP}, n) * (\mathtt{cid} \rightarrowtail (\mathbf{end}, n)) \\
\mathsf{envElimMyPush}(p) \overset{\text{def}}{=} I' \wedge \mathsf{elimPush}(p, \mathtt{cid}, \_) * \mathsf{true} \qquad \mathsf{envElimMyPop}(p) \overset{\text{def}}{=} I' \wedge \mathsf{elimPop}(p, \mathtt{cid}, \_, \_, \_, \_) * \mathsf{true} \\
\mathsf{publish}(p, op) \overset{\text{def}}{=} (I' \wedge \mathsf{notDone}(p, \mathtt{cid}, op, \_) * \mathsf{true}) \vee (\mathsf{envElimMyPush}(p) \wedge op = \mathtt{PUSH}) \vee (\mathsf{envElimMyPop}(p) \wedge op = \mathtt{POP}) \\
\mathsf{hisDesc}(q, op) \overset{\text{def}}{=} \exists n.\, ((\mathtt{loc}[\mathtt{him}] = q) * (\mathtt{him} \rightarrowtail (op, n)) \vee (\mathtt{loc}[\mathtt{him}] \neq q)) * \mathsf{d}(q, \mathtt{him}, op, n) * \mathsf{true} \wedge (q \in \mathtt{D}) \\
\mathsf{toElim}(p, op, q) \overset{\text{def}}{=} \exists n.\, (\mathsf{notInLoc}(p, op, n) \wedge \mathsf{hisDesc}(q, op') \wedge op \neq op') * (\mathtt{cid} \rightarrowtail (op, n)) \\
\mathsf{unfinishedPop}(p, q) \overset{\text{def}}{=} \exists n'.\, (\mathsf{notInLoc}(p, \mathtt{POP}, \_) \wedge \mathsf{d}(q, \_, \mathtt{PUSH}, n') * \mathsf{true}) * (\mathtt{cid} \rightarrowtail (\mathbf{end}, n'))
\end{array}
$$

Figure 27. Precise Invariant, Rely/Guarantee and Auxiliary Definitions of HSY Stack (for Thread t)

Frame out: $\mathsf{stkInv} * \mathsf{collInv}$

```
TryCollision(ThrdInfo p, q)   local b;
  {$\exists op.\, \mathsf{toElim}(\mathtt{p}, op, \mathtt{q})$}
  if (p.op = PUSH) {
    {$\mathsf{toElim}(\mathtt{p}, \mathtt{PUSH}, \mathtt{q})$}
    < b := cas(&loc[q.id], q, p);
      if (b) { lin(p.id); lin(q.id); } >
    {$(\mathtt{b} \wedge \mathsf{endPush}(\mathtt{p})) \vee (\neg\mathtt{b} \wedge \mathsf{toElim}(\mathtt{p}, \mathtt{PUSH}, \mathtt{q}))$}
  } else if (p.op = POP) {
    {$\mathsf{toElim}(\mathtt{p}, \mathtt{POP}, \mathtt{q})$}
    < b := cas(&loc[q.id], q, null);
      if (b) { EPush := EPush $\cup$ {q.id};
               lin(q.id); lin(p.id); } >
    {$(\mathtt{b} \wedge \mathsf{unfinishedPop}(\mathtt{p}, \mathtt{q})) \vee (\neg\mathtt{b} \wedge \mathsf{toElim}(\mathtt{p}, \mathtt{POP}, \mathtt{q}))$}
    if (b) p.data := q.data;
    {$(\mathtt{b} \wedge \mathsf{endPop}(\mathtt{p})) \vee (\neg\mathtt{b} \wedge \mathsf{toElim}(\mathtt{p}, \mathtt{POP}, \mathtt{q}))$}
  }
  { $(\mathtt{b} \wedge (\mathsf{endPush}(\mathtt{p}) \vee \mathsf{endPop}(\mathtt{p}))) \vee (\neg\mathtt{b} \wedge \exists op.\, \mathsf{toElim}(\mathtt{p}, op, \mathtt{q}))$ }
```

Figure 28. Proof Outline of `TryCollision` in HSY Stack

Frame out: $\mathsf{stkInv} * \mathsf{collInv}$

```
void FinishCollision(ThrdInfo p)
  {$\mathsf{envElimMyPush}(\mathtt{p}) \vee \mathsf{envElimMyPop}(\mathtt{p})$}
   if (p.op = POP) {
     {$\mathsf{envElimMyPop}(\mathtt{p})$}
      p.data := loc[cid].data;
     {$\exists n.\, I' \wedge \mathsf{elimPop}(p, \mathtt{cid}, n, \_, \_, n) * \mathsf{true}$}
      loc[cid] := null;
     {$\mathsf{endPop}(\mathtt{p})$}
   } else if (p.op = PUSH) {
     {$\mathsf{envElimMyPush}(\mathtt{p})$}
      EPush := EPush \ {cid};
     {$\mathsf{endPush}(\mathtt{p})$}
   }
  {$\mathsf{endPush}(\mathtt{p}) \vee \mathsf{endPop}(\mathtt{p})$}
```

Figure 29. Proof Outline of `FinishCollision` in HSY Stack

operation and the corresponding code for a pop operation. We insert **lin**(`p.id`) and **lin**(`q.id`) at LPs, to linearize the threads `p.id` (which is `cid`) and `q.id` (which is `him`). We give auxiliary definitions in Fig. 27. The precondition says that, before the elimination, the current thread has not done its operation ($\mathtt{cid} \rightarrowtail (op, n)$),

```
Frame out: stkInv

IntSet GN;    //Aux: popped garbage nodes
IntSet D;   //Aux: all thread descriptors (added into D when allocated)
IntSet EPush;   //Aux: eliminated pushes

void StackOp(ThrdInfo p)
  local him, q, pos, r;
 {∃op. begin(p, op)}
  while(true) {
    if (TryStackOp(p))
      {endPush(p) ∨ endPop(p)}
      return;
    {∃op. begin(p, op)}
    loc[cid] := p;
    {∃op. publish(p, op)}
    pos := GetPosition(p);
    him := coll[pos];
    while (!cas(&coll[pos], him, cid))
      him := coll[pos];
    if (1 <= him <= thrdNum) {
      q := loc[him];
      if (q != null && q.id = him && q.op != p.op) {
       {∃op. publish(p, op) ∧ hisDesc(q, op′) ∧ op ≠ op′}
        if (cas(&loc[cid], p, null)) {
         {∃op. toElim(p, op, q)}
          if (TryCollision(p, q))
           {endPush(p) ∨ endPop(p)}
            return;
          else
           {∃op. begin(p, op)}
            continue;
        } else {
         {envElimMyPush(p) ∨ envElimMyPop(p)}
          FinishCollision(p);
         {endPush(p) ∨ endPop(p)}
          return; }
      }
    }
    {∃op. publish(p, op)}
    if (!cas(&loc[cid], p, null)) {
      {envElimMyPush(p) ∨ envElimMyPop(p)}
      FinishCollision(p);
      {endPush(p) ∨ endPop(p)}
      return;
    }
    {∃op. begin(p, op)}
  }
```

Figure 30. Proof Outline of `StackOp` in HSY Stack (Thread `cid`)

its descriptor `p` is not in the `loc` array (notInLoc) and it knows the descriptor `q` (hisDesc) which holds an opposite operation. The postcondition says that, if the elimination is successful (b holds), the current thread has done its operation (endPush or endPop). In the proof, we can frame out the central stack and the collision array by the FRAME rule. Abstract operations as auxiliary states are no different from normal states. Since the algorithm does not have future-dependent LPs, we do not need speculation.

E.3 MS Two-Lock Queue

Michael and Scott's two-lock queue [22] uses a linked list with `Head` and `Tail` pointers to implement a queue. We show the concrete implementation in Figure 31(a). The list always contain a sentinel node (it is allocated when constructing a new queue, as shown in the `initialize` method). Enqueue operates on the tail of the queue, while dequeue operates at the head of the queue, which always replaces the sentinel node by its next node and returns the value in the new sentinel node. The concrete queue is protected by two locks, `Hlock` and `Tlock`. They ensure that at most one enqueue thread and one dequeue thread at a time can access the queue. Also, since we have two locks, an enqueue thread do not need to wait for a dequeue thread, or vice versa.

```
struct Node {
  int val;
  struct Node *next;
}
struct Queue {
  struct Node *Head;
  struct Node *Tail;
  int Hlock;
  int Tlock;
}
initialize(){
  local dummy;
  dummy := cons(0, null);
  Head := dummy;
  Tail := dummy;
  Hlock := 0;
  Tlock := 0;
}

enq(v) :
     local x;
  1  x := cons(v, null);
  2  lock(Tlock);
  3  Tail.next := x;
  4  Tail := x;
  5  unlock(Tlock);

int deq() :
     local h, s, v;
  6  lock(Hlock);
  7  h := Head;
  8  s := h.next;
  9  if (s = null) {
 10    unlock(Hlock);
 11    return EMPTY;
 12  }
 13  v := s.val;
 14  Head := s;
 15  unlock(Hlock);
 16  return v;
```

(a) Concrete Implementation

$$\theta \in \{\mathtt{Q}\} \to \mathit{List}(\mathit{Int})$$

$$\mathrm{ENQ}(v)(\theta) \overset{\text{def}}{=} (\mathbf{void}, \theta\{\mathtt{Q} \rightsquigarrow \theta(\mathtt{Q}) :: v\})$$

$$\mathrm{DEQ}(\_)(\theta) \overset{\text{def}}{=} \begin{cases} (v, \theta\{\mathtt{Q} \rightsquigarrow q\}) & \text{if } \theta(\mathtt{Q}) = v :: q \\ \mathtt{EMPTY} & \text{otherwise} \end{cases}$$

(b) Abstract Operations

Figure 31. MS Two-Lock Queue Code

As well as the stack in previous sections, we represent an abstract queue `Q` by a sequence of values. The abstract ENQ and DEQ operations, as defined in Figure 31, append values at the end of the abstract queue and remove the first value in the sequence respectively. When the queue is empty, we assume DEQ returns `EMPTY`.

We define the precise invariant, the rely and the guarantee in Figure 32, and show the proof in Figure 33, where we highlight the instrumented auxiliary commands.

The invariant I in Figure 32 maps the concrete list to the abstract queue. Intuitively, the value sequence in the concrete list should be the same as the abstract side, but sometimes the situations are trickier:

$$
\begin{array}{l}
I \overset{\text{def}}{=} \exists A.\, (\mathsf{unlagq}(A) \lor \mathsf{lagq}(\_, A) \lor \mathsf{cross}(A)) * (\mathtt{Q} \Mapsto A) * \mathsf{garb} \\
\mathsf{node}(x, v, y) \overset{\text{def}}{=} x \mapsto (v, y) \qquad \mathsf{node}(x, y) \overset{\text{def}}{=} \mathsf{node}(x, \_, y) \qquad \mathsf{garb} \overset{\text{def}}{=} \circledast_{x \in \mathtt{GN}}.\mathsf{node}(x, \_) \\
\mathsf{last2}(t, v, x, v') \overset{\text{def}}{=} \mathsf{node}(t, v, x) * \mathsf{node}(x, v', \mathtt{null}) \qquad \mathsf{last2}(t, x) \overset{\text{def}}{=} \exists v, v'.\, \mathsf{last2}(t, v, x, v') \\
\mathsf{ls}(x, A, y) \overset{\text{def}}{=} (x = y \land A = \epsilon) \lor (x \neq y \land \exists z, v, A'.\, A = v :: A' \land \mathsf{node}(x, v, z) * \mathsf{ls}(z, A', y)) \qquad \mathsf{ls}(x, y) \overset{\text{def}}{=} \exists A.\, \mathsf{ls}(x, A, y) \\
\mathsf{unlagq}(A) \overset{\text{def}}{=} \exists v_d, v, A'.\, (v_d :: A = A' :: v) \land \mathsf{ls}(\mathtt{Head}, A', \mathtt{Tail}) * \mathsf{node}(\mathtt{Tail}, v, \mathtt{null}) * (\mathtt{Hlock} = \_) * (\mathtt{Tlock} = \_) \\
\mathsf{lagq}(x, A) \overset{\text{def}}{=} \exists v_d, v, v', A'.\, (v_d :: A = A' :: v :: v') \land \mathsf{ls}(\mathtt{Head}, A', \mathtt{Tail}) * \mathsf{last2}(\mathtt{Tail}, v, x, v') * (\mathtt{Hlock} = \_) * (\mathtt{Tlock} \neq 0) \\
\mathsf{cross}(A) \overset{\text{def}}{=} (A = \epsilon) \land \mathsf{node}(\mathtt{Tail}, \mathtt{Head}) * \mathsf{node}(\mathtt{Head}, \mathtt{null}) * (\mathtt{Hlock} = \_) * (\mathtt{Tlock} \neq 0) \qquad \mathsf{cross} \overset{\text{def}}{=} \exists A.\, \mathsf{cross}(A)
\end{array}
$$

$$
\begin{array}{l}
R_{\mathsf{t}} \overset{\text{def}}{=} \bigvee_{\mathsf{t}' \neq \mathsf{t}} G_{\mathsf{t}'} \\
G_{\mathsf{t}} \overset{\text{def}}{=} [\mathsf{Enq}_{\mathsf{t}} \lor \mathsf{Deq}_{\mathsf{t}} \lor \mathsf{Swing}_{\mathsf{t}} \lor \mathsf{LockH}_{\mathsf{t}} \lor \mathsf{UnlockH}_{\mathsf{t}} \lor \mathsf{LockT}_{\mathsf{t}} \lor \mathsf{UnlockT}_{\mathsf{t}}]_I \\
\mathsf{Enq}_{\mathsf{t}} \overset{\text{def}}{=} \exists v, v', A.\, ((\mathtt{Tlock} = \mathtt{t}) * \mathsf{node}(\mathtt{Tail}, v, \mathtt{null}) * \mathtt{Q} \Mapsto A) \ltimes ((\mathtt{Tlock} = \mathtt{t}) * \mathsf{last2}(\mathtt{Tail}, v, \_, v') * \mathtt{Q} \Mapsto A :: v') \\
\mathsf{Deq}_{\mathsf{t}} \overset{\text{def}}{=} \exists v, A, x, y, z, S.\, ((\mathtt{Hlock} = \mathtt{t}) * (\mathtt{Head} = x) \land \mathsf{node}(x, y) * \mathsf{node}(y, v, z) * (\mathtt{GN} = S) * \mathtt{Q} \Mapsto v :: A) \\
\qquad \ltimes ((\mathtt{Hlock} = \mathtt{t}) * (\mathtt{Head} = y) \land \mathsf{node}(x, y) * \mathsf{node}(y, z) * (\mathtt{GN} = S \cup \{x\}) * \mathtt{Q} \Mapsto A) \\
\mathsf{Swing}_{\mathsf{t}} \overset{\text{def}}{=} \exists v, v', x, y.\, ((\mathtt{Tlock} = \mathtt{t}) * (\mathtt{Tail} = x) \land \mathsf{last2}(x, v, y, v')) \ltimes ((\mathtt{Tlock} = \mathtt{t}) * (\mathtt{Tail} = y) \land \mathsf{last2}(x, v, y, v')) \\
\mathsf{LockH}_{\mathsf{t}} \overset{\text{def}}{=} (\mathtt{Hlock} = 0) \ltimes (\mathtt{Hlock} = \mathtt{t}) \qquad \mathsf{UnlockH}_{\mathsf{t}} \overset{\text{def}}{=} (\mathtt{Hlock} = \mathtt{t}) \ltimes (\mathtt{Hlock} = 0) \\
\mathsf{LockT}_{\mathsf{t}} \overset{\text{def}}{=} (\mathtt{Tlock} = 0) \ltimes (\mathtt{Tlock} = \mathtt{t}) \qquad \mathsf{LockT}_{\mathsf{t}} \overset{\text{def}}{=} (\mathtt{Tlock} = \mathtt{t}) \ltimes (\mathtt{Tlock} = 0)
\end{array}
$$

Figure 32. Precise Invariant, Rely and Guarantee of MS Two-Lock Queue (for Thread t)

1. The enq method first appends the new node to the list and then update the Tail pointer. This means, Tail may lag behind the end of the list. Nevertheless, Tail always points to either the last node or a node pointing to the last node in the list.
2. Starting from an empty list, when the deq operation happens in the middle of the enq operation, the Tail may point to an old sentinel node which have been dequeued. In this case, Head and Tail will "cross": Tail points to a node, whose next node is pointed to by Head. The queue is "empty" at that time (it only contains the new sentinel node).

Thus in the invariant I, we distinguish three cases: unlagq describes the concrete queue when Tail does not lag behind (*i.e.*, Tail's next is null), lagq specifies a non-empty queue whose Tail has not yet swung to the end, and cross is for the queue when Head and Tail cross. The value sequence A should not contain the value of the sentinel node, but should contain the new end node even if Tail lags behind. It corresponds to the abstract queue Q. Also, as in the stack algorithms, the precise invariant I contains the garbage nodes (garb). We introduce the write-only auxiliary variable GN to remember those nodes which were once on the queue but have been dequeued.

The guarantee defined in Figure 32 allows a thread to require and release the Hlock and Tlock locks (LockH, UnlockH, LockT and UnlockT), to enqueue a node at Tail when holding the Tlock lock (Enq), to swing the Tail pointer to the end node (Swing), and to dequeue a node at Head when holding the Head lock (Deq).

The linearization points of the implementation are at line 3 for ENQ, line 8 for DEQ from an empty queue, and line 14 for DEQ from a non-empty queue. They do not involve helping mechanism or depend on future executions.

The proof shown in Figure 33 follows the rely-guarantee reasoning. We need to make sure an assertion at each program point is stable *w.r.t.* the environment actions. In particular, for the enq method, we need to consider possible Deq actions from the environment. Thus before line 4, the Tail pointer lags behind and it may also cross with the Head pointer. Similarly, for the deq method, we need to take into account possible Enq and Swing actions from the environment.

E.4 MS Lock-Free Queue

In addition to the two-lock queue, Michael and Scott also propose a lock-free queue [22]. We have shown its code in Figure 15.

We have discussed the instrumentation in Sec. 6.2. Here we define the precise invariant, the rely and the guarantee in Figure 34, and show the proof in Figures 35 and 36, where we highlight the instrumented auxiliary commands.

The invariant I in Figure 34 maps the value sequence in the concrete list to the abstract queue Q. As in the MS two-lock queue, we need to consider the case when the Tail pointer lags behind the end of the list. But here, Head and Tail will never cross, because in this algorithm, a thread will dequeue a node only when Head does not equal Tail. First, the check at line 22 in Figure 15 compares h and t which was read from Tail. If they are not equal, we know h cannot be equal to the current Tail. Besides, at line 28, a node is dequeued only when h is still Head. Thus at line 28 when swinging Head to the next node, Head must be different from Tail. This means, Head will never go "faster" than Tail, and they will never cross. As usual, I still contains the garbage nodes (garb). We use the auxiliary variable GN to collect those nodes which were dequeued from the list.

The guarantee G defined in Figure 34 contains three actions: enqueue, dequeue and swing the Tail pointer. Their definitions are very similar to those for the two-lock queue in Appendix E.3, but without locks. An important difference is that, here Deq requires Head is not equal to Tail before the action. This is the key to ensuring that Head and Tail will not cross, as we discussed.

The proofs in Figures 35 and 36 follow the intuition of the algorithm. Our logic and code instrumentation provide a powerful technique to express and reason about speculation.

We also verify a variant of the deq method which does not need speculation. The code with the proof is shown in Figure 37, where we just remove the rechecking at line 21 in Figure 15. Then, line 20 is a fixed linearization point. We insert **linself** at line 20 and do not need commit anymore. The proof is very similar to the original deq's proof, which confirms our understanding that the rechecking is an optimization, and does not affect the correctness of the algorithm.

$$I_{\mathsf{unlag}}(t) \stackrel{\text{def}}{=} \exists A.\, \mathsf{unlagq}(A) * (\mathtt{Q} \Mapsto A) * \mathsf{garb} \wedge (\mathtt{Tlock} = t) * \mathsf{true}$$
$$I_{\mathsf{lag}}(t, x) \stackrel{\text{def}}{=} \exists A.\, (\mathsf{lagq}(x, A) \vee (\mathsf{cross}(A) \wedge x = \mathtt{Head})) * (\mathtt{Q} \Mapsto A) * \mathsf{garb} \wedge (\mathtt{Tlock} = t) * \mathsf{true}$$

```
enq(v):
  local x;
  $\{ I * \mathtt{cid} \rightarrowtail (\mathrm{ENQ}, \mathtt{v}) \}$
  x := cons(v, null);
  $\{ I * \mathsf{node}(\mathtt{x}, \mathtt{v}, \mathtt{null}) * \mathtt{cid} \rightarrowtail (\mathrm{ENQ}, \mathtt{v}) \}$
  lock(Tlock);
  $\{ I_{\mathsf{unlag}}(\mathtt{cid}) * \mathsf{node}(\mathtt{x}, \mathtt{v}, \mathtt{null}) * \mathtt{cid} \rightarrowtail (\mathrm{ENQ}, \mathtt{v}) \}$
  < Tail.next := x; linself; >
  $\{ I_{\mathsf{lag}}(\mathtt{cid}, \mathtt{x}) * \mathtt{cid} \rightarrowtail (\mathbf{end}, \mathbf{void}) \}$
  Tail := x;
  $\{ (I_{\mathsf{unlag}}(\mathtt{cid}) \wedge (\mathtt{Tail} = \mathtt{x}) * \mathsf{true}) * \mathtt{cid} \rightarrowtail (\mathbf{end}, \mathbf{void}) \}$
  unlock(Tlock);
  $\{ I * \mathtt{cid} \rightarrowtail (\mathbf{end}, \mathbf{void}) \}$
```

$$\mathsf{readheadnext\_null}(s) \stackrel{\text{def}}{=} (s = \mathtt{null}) * (\mathsf{cross} \vee \mathsf{ls}(\mathtt{Head}, \mathtt{Tail})) * \mathsf{true}$$
$$\mathsf{readheadnext\_notnull}(s) \stackrel{\text{def}}{=} (\mathsf{node}(\mathtt{Head}, s) * \mathsf{ls}(s, \mathtt{Tail}) * \mathsf{true}) \vee ((\mathtt{Head} = \mathtt{Tail}) * \mathsf{node}(\mathtt{Head}, s) * \mathsf{node}(s, \mathtt{null}) * \mathsf{true})$$
$$\mathsf{readval}(s, v) \stackrel{\text{def}}{=} (s = \mathtt{Tail} \wedge (\mathsf{node}(s, v, \mathtt{null}) \vee \mathsf{last2}(s, v, \_, \_))) \vee (s \neq \mathtt{Tail} \wedge \exists x.\, \mathsf{node}(s, v, x) * \mathsf{ls}(x, \mathtt{Tail}))$$
$$\mathsf{readnextval}(h, s, v) \stackrel{\text{def}}{=} \mathsf{node}(h, s) * (\mathsf{readval}(s, v) \vee (\mathtt{Tail} = h) * \mathsf{node}(s, v, \mathtt{null}))$$

```
IntSet GN;   //Auxiliary global variable for verification: dequeued nodes

deq():
  local h, s, v;
  $\{ I * \mathtt{cid} \rightarrowtail \mathrm{DEQ} \}$
  lock(Hlock);
  $\{ (I \wedge (\mathtt{Hlock} = \mathtt{t}) * \mathsf{true}) * \mathtt{cid} \rightarrowtail \mathrm{DEQ} \}$
  h := Head;
  $\{ (I \wedge ((\mathtt{Hlock} = \mathtt{t}) \wedge (\mathtt{h} = \mathtt{Head})) * \mathsf{true}) * \mathtt{cid} \rightarrowtail \mathrm{DEQ} \}$
  < s := h.next; if (s = null) linself; >
  $\{ (I \wedge ((\mathtt{Hlock} = \mathtt{t}) \wedge (\mathtt{h} = \mathtt{Head})) * \mathsf{true} \wedge \mathsf{readheadnext\_null}(s)) * \mathtt{cid} \rightarrowtail (\mathbf{end}, \mathrm{EMPTY})$
  $\quad \vee (I \wedge ((\mathtt{Hlock} = \mathtt{t}) \wedge (\mathtt{h} = \mathtt{Head})) * \mathsf{true} \wedge \mathsf{readheadnext\_notnull}(s)) * \mathtt{cid} \rightarrowtail \mathrm{DEQ} \}$
  if (s = null) {
    $\{ (I \wedge ((\mathtt{Hlock} = \mathtt{t}) \wedge (\mathtt{h} = \mathtt{Head})) * \mathsf{true} \wedge \mathsf{readheadnext\_null}(s)) * \mathtt{cid} \rightarrowtail (\mathbf{end}, \mathrm{EMPTY}) \}$
    unlock(Hlock);
    $\{ I * \mathtt{cid} \rightarrowtail (\mathbf{end}, \mathrm{EMPTY}) \}$
    return EMPTY;
  }
  $\{ (I \wedge ((\mathtt{Hlock} = \mathtt{t}) \wedge (\mathtt{h} = \mathtt{Head})) * \mathsf{true} \wedge \mathsf{readheadnext\_notnull}(s)) * \mathtt{cid} \rightarrowtail \mathrm{DEQ} \}$
  v := s.val;
  $\{ (I \wedge ((\mathtt{Hlock} = \mathtt{t}) \wedge (\mathtt{h} = \mathtt{Head})) * \mathsf{true} \wedge \mathsf{readnextval}(\mathtt{Head}, \mathtt{s}, \mathtt{v}) * \mathsf{true}) * \mathtt{cid} \rightarrowtail \mathrm{DEQ} \}$
  < Head := s;  GN := GN ∪ {h};  linself; >
  $\{ (I \wedge (\mathtt{Hlock} = \mathtt{t}) * \mathsf{true}) * \mathtt{cid} \rightarrowtail (\mathbf{end}, \mathtt{v}) \}$
  unlock(Hlock);
  $\{ I * \mathtt{cid} \rightarrowtail (\mathbf{end}, \mathtt{v}) \}$
  return v;
```

Figure 33. Proof Outline of MS Two-Lock Queue for Thread `cid`

$$I \overset{\text{def}}{=} \exists A.\ \mathsf{lsq}(A) * \mathtt{Q} \Rrightarrow A * \mathsf{garb}$$

$$\mathsf{node}(x, v, y) \overset{\text{def}}{=} x \mapsto (v, y) \qquad \mathsf{node}(x, y) \overset{\text{def}}{=} \mathsf{node}(x, \_, y) \qquad \mathsf{garb} \overset{\text{def}}{=} \circledast_{x \in \mathtt{GN}}.\mathsf{node}(x, \_)$$

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

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

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

$$\mathsf{lsq}(A) \overset{\text{def}}{=} \exists v, A', A''.\ (v :: A = A' :: A'') \wedge \mathsf{ls}(\mathtt{Head}, A', \mathtt{Tail}) * \mathsf{tails}(\mathtt{Tail}, \_, A'')$$

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

$$\mathsf{Enq} \overset{\text{def}}{=} \exists v, v', A, x.\ (\mathsf{node}(\mathtt{Tail}, v, \mathtt{null}) * \mathtt{Q} \Rrightarrow A) \ltimes (\mathsf{last2}(\mathtt{Tail}, v, x, v') * \mathtt{Q} \Rrightarrow A :: v')$$

$$\mathsf{Deq} \overset{\text{def}}{=} \exists v, A, x, y, z, S.\ (\mathtt{Head} = x \wedge x \neq \mathtt{Tail} \wedge \mathsf{node}(x, y) * \mathsf{node}(y, v, z) * (\mathtt{GN} = S) * \mathtt{Q} \Rrightarrow v :: A)$$
$$\ltimes (\mathtt{Head} = y \wedge \mathsf{node}(x, y) * \mathsf{node}(y, z) * (\mathtt{GN} = S \cup \{x\}) * \mathtt{Q} \Rrightarrow A)$$

$$\mathsf{Swing} \overset{\text{def}}{=} \exists v, v', x, y.\ (\mathtt{Tail} = x \wedge \mathsf{last2}(x, v, y, v')) \ltimes (\mathtt{Tail} = y \wedge \mathsf{last2}(x, v, y, v'))$$

Figure 34. Precise Invariant, Rely and Guarantee of MS Lock-Free Queue

$$\mathsf{readnext\_envenq}(t, s) \overset{\text{def}}{=} (s = \mathtt{null}) \wedge \exists x.\ \mathsf{node}(t, x) * \mathsf{node}(x, \_)$$

$$\mathsf{readtailnext}(t, s) \overset{\text{def}}{=} (t = \mathtt{Tail} \wedge \mathsf{tails}(t, s)) \vee (t \neq \mathtt{Tail} \wedge \mathsf{node}(t, s) * \mathsf{ls}(s, \mathtt{Tail})) \vee \mathsf{readnext\_envenq}(t, s)$$

```
enq(v):
    local x, t, s, r;
   $\{ I * \mathtt{cid} \rightarrowtail (\mathrm{ENQ}, \mathrm{v}) \}$
    x := cons(v, null);
   $\{ I * \mathsf{node}(\mathrm{x}, \mathrm{v}, \mathtt{null}) * \mathtt{cid} \rightarrowtail (\mathrm{ENQ}, \mathrm{v}) \}$
    while (true) {
      t := Tail;
     $\{ (I \wedge \mathsf{ls}(\mathrm{t}, \mathtt{Tail}) * \mathsf{true}) * \mathsf{node}(\mathrm{x}, \mathrm{v}, \mathtt{null}) * \mathtt{cid} \rightarrowtail (\mathrm{ENQ}, \mathrm{v}) \}$
      s := t.next;
     $\{ (I \wedge \mathsf{readtailnext}(\mathrm{t}, \mathrm{s}) * \mathsf{true}) * \mathsf{node}(\mathrm{x}, \mathrm{v}, \mathtt{null}) * \mathtt{cid} \rightarrowtail (\mathrm{ENQ}, \mathrm{v}) \}$
      if (t = Tail) {
       $\{ (I \wedge \mathsf{readtailnext}(\mathrm{t}, \mathrm{s}) * \mathsf{true}) * \mathsf{node}(\mathrm{x}, \mathrm{v}, \mathtt{null}) * \mathtt{cid} \rightarrowtail (\mathrm{ENQ}, \mathrm{v}) \}$
        if (s = null) {
         $\{ (I \wedge ((\mathrm{t} = \mathtt{Tail} \wedge \mathsf{node}(\mathrm{t}, \mathrm{s}) \wedge \mathrm{s} = \mathtt{null}) \vee \mathsf{readnext\_envenq}(\mathrm{t}, \mathrm{s})) * \mathsf{true}) * \mathsf{node}(\mathrm{x}, \mathrm{v}, \mathtt{null}) * \mathtt{cid} \rightarrowtail (\mathrm{ENQ}, \mathrm{v}) \}$
          < r := cas(&(t.next), s, x);  if (r) linself; >
         $\{ \mathrm{r} = 0 * I * \mathsf{node}(\mathrm{x}, \mathrm{v}, \mathtt{null}) * \mathtt{cid} \rightarrowtail (\mathrm{ENQ}, \mathrm{v})$
           $\vee\, \mathrm{r} = 1 * (I \wedge (\mathrm{t} = \mathtt{Tail} \Rightarrow \mathsf{last2}(\mathrm{t}, \mathrm{x})) * \mathsf{true}) * \mathtt{cid} \rightarrowtail (\mathbf{end}, \mathbf{void}) \}$
          if (r) {
           $\{ \mathrm{r} = 1 * (I \wedge (\mathrm{t} = \mathtt{Tail} \Rightarrow \mathsf{last2}(\mathrm{t}, \mathrm{x})) * \mathsf{true}) * \mathtt{cid} \rightarrowtail (\mathbf{end}, \mathbf{void}) \}$
            cas(&Tail, t, x);
           $\{ \mathrm{r} = 1 * I * \mathtt{cid} \rightarrowtail (\mathbf{end}, \mathbf{void}) \}$
            return;
          }
         $\{ \mathrm{r} = 0 * I * \mathsf{node}(\mathrm{x}, \mathrm{v}, \mathtt{null}) * \mathtt{cid} \rightarrowtail (\mathrm{ENQ}, \mathrm{v}) \}$
        } else
         $\{ (I \wedge (\mathrm{t} = \mathtt{Tail} \Rightarrow \mathsf{last2}(\mathrm{t}, \mathrm{s})) * \mathsf{true}) * \mathsf{node}(\mathrm{x}, \mathrm{v}, \mathtt{null}) * \mathtt{cid} \rightarrowtail (\mathrm{ENQ}, \mathrm{v}) \}$
          cas(&Tail, t, s);
         $\{ I * \mathsf{node}(\mathrm{x}, \mathrm{v}, \mathtt{null}) * \mathtt{cid} \rightarrowtail (\mathrm{ENQ}, \mathrm{v}) \}$
      }
    }
```

Figure 35. Proof Outline for Enqueue of MS Lock-Free Queue for Thread `cid`

$$\mathsf{readheadnext\_aftertail}(h,s,t) \stackrel{\text{def}}{=} (h = t \wedge \mathsf{readtailnext}(t,s)) \vee (h \neq t \wedge \mathsf{node}(h,s) * \mathsf{ls}(s,t) * \mathsf{ls}(t,\mathtt{Tail}))$$
$$\mathsf{readval}(s,v) \stackrel{\text{def}}{=} (s = \mathtt{Tail} \wedge (\mathsf{node}(s,v,\mathtt{null}) \vee \mathsf{last2}(s,v,\_,\_))) \vee (s \neq \mathtt{Tail} \wedge \exists x.\, \mathsf{node}(s,v,x) * \mathsf{ls}(x,\mathtt{Tail}))$$

`IntSet GN;` //Auxiliary global variable for verification: dequeued nodes

```
deq():
    local h, t, s, v, r;
   $\{ I * \mathtt{cid} \rightarrowtail \mathsf{DEQ} \}$
    while (true) {
      h := Head;
     $\{ (I \wedge \mathsf{ls}(\mathtt{h},\mathtt{Head}) * \mathsf{true}) * \mathtt{cid} \rightarrowtail \mathsf{DEQ} \}$
      t := Tail;
     $\{ (I \wedge \mathsf{ls}(\mathtt{h},\mathtt{Head}) * \mathsf{true} \wedge \mathsf{ls}(\mathtt{h},\mathtt{t}) * \mathsf{ls}(\mathtt{t},\mathtt{Tail}) * \mathsf{true}) * \mathtt{cid} \rightarrowtail \mathsf{DEQ} \}$
      < s := h.next; if (h = t && s = null) trylinself; >
     $\left\{ \begin{array}{l} (I \wedge \mathsf{ls}(\mathtt{h},\mathtt{Head}) * \mathsf{true} \wedge \mathsf{readheadnext\_aftertail}(\mathtt{h},\mathtt{s},\mathtt{t}) * \mathsf{true}) \\ \quad * ((\mathtt{h} = \mathtt{t} \wedge \mathtt{s} = \mathtt{null} \wedge (\mathtt{cid} \rightarrowtail \mathsf{DEQ} \oplus \mathtt{cid} \rightarrowtail (\mathbf{end}, \mathsf{EMPTY}))) \vee ((\mathtt{h} \neq \mathtt{t} \vee \mathtt{s} \neq \mathtt{null}) \wedge \mathtt{cid} \rightarrowtail \mathsf{DEQ})) \end{array} \right\}$
      if (h = Head) {
        if (h = t) {
          if (s = null) {
           $\{ I * (\mathtt{h} = \mathtt{t} \wedge \mathtt{s} = \mathtt{null} \wedge (\mathtt{cid} \rightarrowtail \mathsf{DEQ} \oplus \mathtt{cid} \rightarrowtail (\mathbf{end}, \mathsf{EMPTY}))) \}$
            commit(cid $\rightarrowtail$ (end, EMPTY));
           $\{ I * (\mathtt{h} = \mathtt{t} \wedge \mathtt{s} = \mathtt{null} \wedge \mathtt{cid} \rightarrowtail (\mathbf{end}, \mathsf{EMPTY})) \}$
            return EMPTY;
          }
         $\{ (I \wedge \mathtt{h} = \mathtt{t} \wedge \mathtt{s} \neq \mathtt{null} \wedge \mathsf{readtailnext}(\mathtt{t},\mathtt{s}) * \mathsf{true}) * \mathtt{cid} \rightarrowtail \mathsf{DEQ} \}$
         $\{ (I \wedge (\mathtt{t} = \mathtt{Tail} \Rightarrow \mathsf{last2}(\mathtt{t},\mathtt{s})) * \mathsf{true}) * \mathtt{cid} \rightarrowtail \mathsf{DEQ} \}$
          cas(&Tail, t, s);
         $\{ I * \mathtt{cid} \rightarrowtail \mathsf{DEQ} \}$
        } else {
         $\{ (I \wedge \mathtt{h} \neq \mathtt{t} \wedge \mathsf{node}(\mathtt{h},\mathtt{s}) * \mathsf{ls}(\mathtt{s},\mathtt{Tail}) * \mathsf{true}) * \mathtt{cid} \rightarrowtail \mathsf{DEQ} \}$
          v := s.val;
         $\{ (I \wedge \mathsf{node}(\mathtt{h},\mathtt{s}) * \mathsf{readval}(\mathtt{s},\mathtt{v}) * \mathsf{true}) * \mathtt{cid} \rightarrowtail \mathsf{DEQ} \}$
          < r := cas(&Head, h, s);  GN := GN $\cup$ {h};  if (r) linself; >
         $\left\{ \begin{array}{l} \mathtt{r} = 0 * I * \mathtt{cid} \rightarrowtail \mathsf{DEQ} \\ \ \vee \mathtt{r} = 1 * I * \mathtt{cid} \rightarrowtail (\mathbf{end}, \mathtt{v}) \end{array} \right\}$
          if (r)
            return v;
        }
      } else {
        commit(cid $\rightarrowtail$ DEQ);
       $\{ I * \mathtt{cid} \rightarrowtail \mathsf{DEQ} \}$
      }
    }
```

Figure 36. Proof Outline for Dequeue of MS Lock-Free Queue for Thread `cid`

```
IntSet GN;   //Auxiliary global variable for verification: dequeued nodes

deq_without_rechecking():
    local h, t, s, v, r;
   $\{ I * \mathsf{cid} \rightarrowtail \mathsf{DEQ} \}$
    while (true) {
      h := Head;
     $\{ (I \wedge \mathsf{ls}(\mathtt{h}, \mathtt{Head}) * \mathsf{true}) * \mathsf{cid} \rightarrowtail \mathsf{DEQ} \}$
      t := Tail;
     $\{ (I \wedge \mathsf{ls}(\mathtt{h}, \mathtt{Head}) * \mathsf{true} \wedge \mathsf{ls}(\mathtt{h}, \mathtt{t}) * \mathsf{ls}(\mathtt{t}, \mathtt{Tail}) * \mathsf{true}) * \mathsf{cid} \rightarrowtail \mathsf{DEQ} \}$
      < s := h.next;  if (h = t && s = null) linself; >
     $\left\{ \begin{array}{l} (I \wedge \mathsf{ls}(\mathtt{h}, \mathtt{Head}) * \mathsf{true} \wedge \mathsf{readheadnext\_aftertail}(\mathtt{h}, \mathtt{s}, \mathtt{t}) * \mathsf{true}) \\ \quad * ((\mathtt{h} = \mathtt{t} \wedge \mathtt{s} = \mathtt{null} \wedge \mathsf{cid} \rightarrowtail (\mathbf{end}, \mathsf{EMPTY})) \vee ((\mathtt{h} \neq \mathtt{t} \vee \mathtt{s} \neq \mathtt{null}) \wedge \mathsf{cid} \rightarrowtail \mathsf{DEQ})) \end{array} \right\}$
      if (h = t) {
        if (s = null) {
         $\{ I * (\mathtt{h} = \mathtt{t} \wedge \mathtt{s} = \mathtt{null} \wedge \mathsf{cid} \rightarrowtail (\mathbf{end}, \mathsf{EMPTY})) \}$
          return EMPTY;
        }
       $\{ (I \wedge \mathtt{h} = \mathtt{t} \wedge \mathtt{s} \neq \mathtt{null} \wedge \mathsf{readtailnext}(\mathtt{t}, \mathtt{s}) * \mathsf{true}) * \mathsf{cid} \rightarrowtail \mathsf{DEQ} \}$
       $\{ (I \wedge (\mathtt{t} = \mathtt{Tail} \Rightarrow \mathsf{last2}(\mathtt{t}, \mathtt{s})) * \mathsf{true}) * \mathsf{cid} \rightarrowtail \mathsf{DEQ} \}$
        cas(&Tail, t, s);
       $\{ I * \mathsf{cid} \rightarrowtail \mathsf{DEQ} \}$
      } else {
       $\{ (I \wedge \mathtt{h} \neq \mathtt{t} \wedge \mathsf{node}(\mathtt{h}, \mathtt{s}) * \mathsf{ls}(\mathtt{s}, \mathtt{Tail}) * \mathsf{true}) * \mathsf{cid} \rightarrowtail \mathsf{DEQ} \}$
        v := s.val;
       $\{ (I \wedge \mathsf{node}(\mathtt{h}, \mathtt{s}) * \mathsf{readval}(\mathtt{s}, \mathtt{v}) * \mathsf{true}) * \mathsf{cid} \rightarrowtail \mathsf{DEQ} \}$
        < r := cas(&Head, h, s);  GN := GN $\cup$ {h};  if (r) linself; >
       $\left\{ \begin{array}{l} \mathtt{r} = 0 * I * \mathsf{cid} \rightarrowtail \mathsf{DEQ} \\ \quad \vee\, \mathtt{r} = 1 * I * \mathsf{cid} \rightarrowtail (\mathbf{end}, \mathtt{v}) \end{array} \right\}$
        if (r)
          return v;
      }
    }
```

Figure 37. Proof Outline for a Variant of Dequeue in MS Lock-Free Queue (Without Rechecking)

E.5 DGLM Queue

Doherty *et al.* [6] present an optimized version of the deq method in MS lock-free queue, and verify the algorithm by constructing a forward and a backward simulations.

We show the code of their deq method in Figure 38 (the enq method is the same as MS lock-free queue). This new version tests whether Tail points to the sentinel node (line 11 in Figure 38) only after Head has been updated (line 9), while in Michael and Scott's version, the test (line 22 in Figure 15) is performed before knowing the queue is not empty.

In Figure 39, we show the proof for the DGLM queue using our logic. The precise invariant and the rely/guarantee conditions are almost the same as MS lock-free queue. We only show up the differences in Figure 39.

Since now deq allows to dequeue a node when Head equals to Tail, the Head and Tail pointers may cross in some executions. Thus the invariant I should consider the case cross, and the action Deq also needs to be slightly changed.

The linearization points are at similar locations as in MS lock-free queue. The proof of enq is the same and omitted here. Since DGLM queue still rechecks the reads of Head and Tail, the location of a linearization point for the DEQ which returns EMPTY will depend on the future, just like the original MS lock-free queue. Thus we still insert **trylinself** and **commit** to handle the LP.

In the proof, we also need to carefully make sure that the assertion at each program point is stable *w.r.t.* the new environment actions.

```
int deq() :
      local h, t, s, v;
  1   while (true) {
  2     h := Head;
  3     s := h.next;
  4     if (h = Head) {
  5       if (s = null) {
  6         return EMPTY;
  7       }
  8       v := s.val;
  9       if (cas(&Head, h, s)) {
 10         t := Tail;
 11         if (h = t) {
 12           cas(&Tail, t, s);
 13         }
 14         return v;
 15       }
 16     }
 17   }
```

(Only deq is different from MS Lock-Free Queue)

Figure 38. DGLM Queue Code

$I \overset{\text{def}}{=} \exists A.\ (\mathsf{lsq}(A) \vee \mathsf{cross}(A)) * (\mathtt{Q} \Mapsto A) * \mathsf{garb}$ (lsq and garb are the same as MS lock-free queue.)

$\mathsf{cross}(A) \overset{\text{def}}{=} (A = \epsilon) \wedge \mathsf{node}(\mathtt{Tail}, \mathtt{Head}) * \mathsf{node}(\mathtt{Head}, \mathtt{null})$

$R = G \overset{\text{def}}{=} [\mathsf{Enq} \vee \mathsf{Deq} \vee \mathsf{Swing}]_I$ (Enq and Swing are the same as MS lock-free queue.)

$\mathsf{Deq} \overset{\text{def}}{=} \exists v, A, x, y, z, S.\ (\mathtt{Head} = x \wedge \mathsf{node}(x, y) * \mathsf{node}(y, v, z) * (\mathtt{GN} = S) * \mathtt{Q} \Mapsto v :: A)$
$\quad \ltimes (\mathtt{Head} = y \wedge \mathsf{node}(x, y) * \mathsf{node}(y, z) * (\mathtt{GN} = S \cup \{x\}) * \mathtt{Q} \Mapsto A)$ (Not require $\mathtt{Head} \neq \mathtt{Tail}$)

$\mathsf{readnext}(h, s) \overset{\text{def}}{=} \mathsf{node}(h, s) * (s = \mathtt{null} \vee \mathsf{ls}(s, \mathtt{Tail}) \vee (\mathtt{Tail} = h) * \mathsf{node}(s, \mathtt{null}))$

$\mathsf{readheadnext}(h, s) \overset{\text{def}}{=} (h = \mathtt{Head} \wedge \mathsf{readnext}(h, s)) \vee (h \neq \mathtt{Head} \wedge \mathsf{node}(h, s) * \mathsf{ls}(s, \mathtt{Head})) \vee \mathsf{readnext\_envenq}(h, s)$

$\mathsf{readnextval}(h, s, v) \overset{\text{def}}{=} \mathsf{node}(h, s) * (\mathsf{readval}(s, v) \vee (\mathtt{Tail} = h) * \mathsf{node}(s, v, \mathtt{null}))$

```
IntSet GN;   //Auxiliary global variable for verification: dequeued nodes

deq():
    local h, t, s, v, r;
   { $I * \mathtt{cid} \rightarrowtail \mathsf{DEQ}$ }
    while (true) {
      h := Head;
     { $(I \wedge \mathsf{ls}(\mathtt{h}, \mathtt{Head}) * \mathsf{true}) * \mathtt{cid} \rightarrowtail \mathsf{DEQ}$ }
      < s := h.next;  if (h = Head && s = null) trylinself; >
     { $(I \wedge \mathsf{ls}(\mathtt{h}, \mathtt{Head}) * \mathsf{true} \wedge \mathsf{readheadnext}(\mathtt{h}, \mathtt{s}) * \mathsf{true})$
       $* ((\mathtt{s} = \mathtt{null} \wedge (\mathtt{cid} \rightarrowtail \mathsf{DEQ} \oplus \mathtt{cid} \rightarrowtail (\mathbf{end}, \mathsf{EMPTY}))) \vee ((\mathtt{h} \neq \mathtt{Head} \vee \mathtt{s} \neq \mathtt{null}) \wedge \mathtt{cid} \rightarrowtail \mathsf{DEQ}))$ }
      if (h = Head) {
        if (s = NULL) {
         { $I * (\mathtt{s} = \mathtt{null} \wedge (\mathtt{cid} \rightarrowtail \mathsf{DEQ} \oplus \mathtt{cid} \rightarrowtail (\mathbf{end}, \mathsf{EMPTY})))$ }
          commit(cid ↣ (end, EMPTY));
         { $I * (\mathtt{s} = \mathtt{null} \wedge \mathtt{cid} \rightarrowtail (\mathbf{end}, \mathsf{EMPTY}))$ }
          return EMPTY;
        }
       { $(I \wedge \mathsf{readheadnext}(\mathtt{h}, \mathtt{s}) * \mathsf{true}) * (\mathtt{s} \neq \mathtt{null} \wedge \mathtt{cid} \rightarrowtail \mathsf{DEQ})$ }
        v := s.val;
       { $(I \wedge (\mathtt{h} = \mathtt{Head} \Rightarrow \mathsf{readnextval}(\mathtt{h}, \mathtt{s}, \mathtt{v})) * \mathsf{true}) * \mathtt{cid} \rightarrowtail \mathsf{DEQ}$ }
        < r := cas(&Head, h, s);  GN := GN ∪ {h};  if (r) linself; >
       { $((\mathtt{r} = 0) * I * \mathtt{cid} \rightarrowtail \mathsf{DEQ}) \vee ((\mathtt{r} = 1) * (I \wedge \mathsf{node}(\mathtt{h}, \mathtt{s}) * \mathsf{ls}(\mathtt{s}, \mathtt{Head}) * \mathsf{true}) * \mathtt{cid} \rightarrowtail (\mathbf{end}, \mathtt{v}))$ }
        if (r) {
         { $(\mathtt{r} = 1) * (I \wedge \mathsf{node}(\mathtt{h}, \mathtt{s}) * \mathsf{ls}(\mathtt{s}, \mathtt{Head}) * \mathsf{true}) * \mathtt{cid} \rightarrowtail (\mathbf{end}, \mathtt{v})$ }
          t := Tail;
         { $(I \wedge \mathsf{node}(\mathtt{h}, \mathtt{s}) * \mathsf{ls}(\mathtt{s}, \mathtt{Head}) * \mathsf{true} \wedge \mathsf{ls}(\mathtt{t}, \mathtt{Tail}) * \mathsf{true}) * \mathtt{cid} \rightarrowtail (\mathbf{end}, \mathtt{v})$ }
          if (h = t) {
           { $(I \wedge \mathsf{node}(\mathtt{t}, \mathtt{s}) * \mathsf{ls}(\mathtt{s}, \mathtt{Head}) * \mathsf{true} \wedge \mathsf{ls}(\mathtt{t}, \mathtt{Tail}) * \mathsf{true}) * \mathtt{cid} \rightarrowtail (\mathbf{end}, \mathtt{v})$ }
           { $(I \wedge (\mathtt{t} = \mathtt{Tail} \Rightarrow \mathsf{last2}(\mathtt{t}, \mathtt{s})) * \mathsf{true}) * \mathtt{cid} \rightarrowtail (\mathbf{end}, \mathtt{v})$ }
            cas(&Tail, t, s);
           { $I * \mathtt{cid} \rightarrowtail (\mathbf{end}, \mathtt{v})$ }
          }
         { $I * \mathtt{cid} \rightarrowtail (\mathbf{end}, \mathtt{v})$ }
          return v;
        }
      } else {
        commit(cid ↣ DEQ);
       { $I * \mathtt{cid} \rightarrowtail \mathsf{DEQ}$ }
      }
    }
```

Figure 39. Proof for DGLM Queue

```
locate(e):
  local p, c, u;
  $\{ I \wedge (\mathtt{MIN} < \mathtt{e}) \}$
  p := Head;
  lock(p);
  c := p.next;
  u := c.data;
  $\{ \exists v.\ \mathsf{adjacent}(\mathtt{p}, v, \mathtt{c}, \mathtt{u}) \wedge (v < \mathtt{e}) \}$
  while (u < e) {
    lock(c);
    unlock(p);
    p := c;
    c := p.next;
    u := c.data;
  }
  $\{ \exists v, u.\ \mathsf{adjacent}(\mathtt{p}, v, \mathtt{c}, u) \wedge (v < \mathtt{e} \leq u) \}$
  return (p, c);
```

Figure 42. Proof Outline of Locate in Lock-Coupling List

```
add(e):
  local x, y, z, u, r;
  $\{ I * \mathtt{cid} \rightarrowtail (\mathtt{ADD}, \mathtt{e}) \wedge (\mathtt{MIN} < \mathtt{e} < \mathtt{MAX}) \}$
  (x, z) := locate(e);
  $\{ \exists v, u.\ \mathsf{adjacent}(\mathtt{x}, v, \mathtt{z}, u) * \mathtt{cid} \rightarrowtail (\mathtt{ADD}, \mathtt{e})$
  $\quad \wedge (v < \mathtt{e} \leq u) \wedge (\mathtt{e} < \mathtt{MAX}) \}$
  u := z.data;
  if (u != e) {
    $\{ \exists v.\ \mathsf{adjacent}(\mathtt{x}, v, \mathtt{z}, \mathtt{u}) * \mathtt{cid} \rightarrowtail (\mathtt{ADD}, \mathtt{e}) \wedge (v < \mathtt{e} < \mathtt{u}) \}$
    y := cons(0, e, z);
    $\{ \exists v.\ \mathsf{adjacent}(\mathtt{x}, v, \mathtt{z}, \mathtt{u}) * \mathsf{U}(\mathtt{y}, \mathtt{e}, \mathtt{z}) * \mathtt{cid} \rightarrowtail (\mathtt{ADD}, \mathtt{e})$
    $\quad \wedge (v < \mathtt{e} < \mathtt{u}) \}$
    < x.next := y;  linself; >
    $\{ \exists v.\ \mathsf{adjacent}(\mathtt{x}, v, \mathtt{y}, \mathtt{e}) * \mathtt{cid} \rightarrowtail (\mathbf{end}, \mathbf{true}) \}$
    r := true;
  } else {
    $\{ \exists v.\ \mathsf{adjacent}(\mathtt{x}, v, \mathtt{z}, \mathtt{e}) * \mathtt{cid} \rightarrowtail (\mathtt{ADD}, \mathtt{e}) \wedge (\mathtt{e} < \mathtt{MAX}) \}$
    linself;
    $\{ \exists v.\ \mathsf{adjacent}(\mathtt{x}, v, \mathtt{y}, \mathtt{e}) * \mathtt{cid} \rightarrowtail (\mathbf{end}, \mathbf{false}) \}$
    r := false;
  }
  unlock(x);
  $\{ I * \mathtt{cid} \rightarrowtail (\mathbf{end}, \mathtt{r}) \}$
  return r;
```

Figure 43. Proof of Add in Lock-Coupling List (Thread `cid`)

E.6 Lock-Coupling List

Below we will verify the four fine-grained list-based set algorithms in Herlihy and Shavit's book [15]: lock-coupling list, optimistic list, lazy list and lock-free list. In Figure 40(b) we define three abstract set operations, `ADD(e)`, `RMV(e)` and `CTN(e)`, where the abstract set `S` is simply represented by a mathematical set. These abstract operations will serve as the specification for all the four list-based set implementations.

In this section, we verify the lock-coupling list. Figure 40(a) gives its concrete implementation. The abstract set is implemented by an ordered singly-linked list pointed to by a shared variable `Head`, with two sentinel nodes at the two ends of the list containing the values `MIN` and `MAX` 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

```
rmv(e):
  local x, y, z, v;
  $\{ I * \mathtt{cid} \rightarrowtail (\mathtt{RMV}, \mathtt{e}) \wedge (\mathtt{MIN} < \mathtt{e} < \mathtt{MAX}) \}$
  (x, y) := locate(e);
  $\{ \exists u, v.\ \mathsf{adjacent}(\mathtt{x}, u, \mathtt{y}, v) * \mathtt{cid} \rightarrowtail (\mathtt{RMV}, \mathtt{e})$
  $\quad \wedge (u < \mathtt{e} \leq v) \wedge (\mathtt{e} < \mathtt{MAX}) \}$
  v := y.data;
  if (v = e) {
    $\{ \exists u.\ \mathsf{adjacent}(\mathtt{x}, u, \mathtt{y}, \mathtt{e}) * \mathtt{cid} \rightarrowtail (\mathtt{RMV}, \mathtt{e}) \wedge (\mathtt{e} < \mathtt{MAX}) \}$
    lock(y);
    z := y.next;
    $\{ \exists u.\ \mathsf{adjacentLocked}(\mathtt{x}, u, \mathtt{y}, \mathtt{e}, \mathtt{z}) * \mathtt{cid} \rightarrowtail (\mathtt{RMV}, \mathtt{e}) \wedge (\mathtt{e} < \mathtt{MAX}) \}$
    < x.next := z;  linself; >
    unlock(x);
    $\{ I * \mathsf{L}_{\mathtt{cid}}(\mathtt{y}, \mathtt{e}, \mathtt{z}) * \mathtt{cid} \rightarrowtail (\mathbf{end}, \mathbf{true}) \}$
    dispose(y);
    $\{ I * \mathtt{cid} \rightarrowtail (\mathbf{end}, \mathbf{true}) \}$
    return true;
  } else {
    $\{ \exists u.\ \mathsf{adjacent}(\mathtt{x}, u, \mathtt{y}, \mathtt{v}) * \mathtt{cid} \rightarrowtail (\mathtt{RMV}, \mathtt{e}) \wedge (u < \mathtt{e} < \mathtt{v}) \}$
    linself;
    $\{ \exists u.\ \mathsf{adjacent}(\mathtt{x}, u, \mathtt{y}, \mathtt{v}) * \mathtt{cid} \rightarrowtail (\mathbf{end}, \mathbf{false}) \}$
    unlock(x);
    $\{ I * \mathtt{cid} \rightarrowtail (\mathbf{end}, \mathbf{false}) \}$
    return false;
  }
```

Figure 44. Proof of Remove in Lock-Coupling List (Thread `cid`)

predecessor's pointer while both the node to be removed and its predecessor are locked. This implementation does not use helping mechanism or future-dependent LPs.

The linearization point for a successful `add` is at line `17` in Figure 40(a), where the new node is linked onto the list. Similarly, the LP for a successful `rmv` is at line `29` where the node is unlinked from the list. For unsuccessful `add` and `rmv`, the LPs could be at any points when holding the corresponding locks. Here we let them be the points just before releasing the locks at lines `22` and `34`. At these lineariztion points, we simply insert **linself**.

We define the precise invariant, the rely and the guarantee in Figure 41, and show the proofs in Figures 42, 43 and 44, where we highlight the instrumented auxiliary commands.

The invariant I in Figure 41 requires the concrete list should be sorted and its elements constitute the abstract set `S`. Note that every removed node can be explicitly disposed (line `31` in Figure 40) in the lock-coupling list algorithm, since the thread locally owns the node after removing it from the list. Thus we do not need an auxiliary variable to remember those garbage nodes as in previous examples.

The guarantee G defined in Figure 41 contains four actions: locking a node (Lock), releasing the lock of a node (Unlock), adding a node when holding the predecessor's lock (Add), and removing a node when holding both the predecessor's and the node's locks (Rmv). Note that Add and Rmv update both the concrete list and the abstract sets, which correspond to the linearization points. For Rmv, the node disappears from the shared state after it is removed. This means, the node is transferred from the shared memory to the thread's local memory.

The verification which follows our rely-guarantee-style inference rules is not difficult, and we show the proofs in Figures 42, 43 and 44.

```
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);
}
```

```
locate(e) :

  local p, c, u;
  p := Head;
  lock(p);
  c := p.next;
  u := c.data;
  while (u < e) {
    lock(c);
    unlock(p);
    p := c;
    c := p.next;
    u := c.data;
  }
  return (p, c);
```

```
add(e) :

  local x, y, z, u, r;
  (x, z) := locate(e);
  u := z.data;
  if (u != e) {
    y := cons(0, e, z);
    x.next := y;
    r := true;
  } else {
    r := false;
  }
  unlock(x);
  return r;
```

```
rmv(e) :
  local x, y, z, v;
  (x, y) := Head;
  v := y.data;
  if (v = e) {
    lock(y);
    z := y.next;
    x.next := z;
    unlock(x);
    dispose(y);
    return true;
  } else {
    unlock(x);
    return false;
  }
```

(a) Implementation

$$\theta \in \{\mathtt{S}\} \to \textit{Set}(\textit{Int})$$

$$\mathtt{ADD}(n)(\theta) \overset{\text{def}}{=} \begin{cases} (\textbf{true}, \theta\{\mathtt{S} \rightsquigarrow S \cup \{n\}\}) & \text{if } \theta(\mathtt{S}) = S \text{ and } n \notin S \\ (\textbf{false}, \theta) & \text{otherwise} \end{cases}$$

$$\mathtt{RMV}(n)(\theta) \overset{\text{def}}{=} \begin{cases} (\textbf{true}, \theta\{\mathtt{S} \rightsquigarrow S\}) & \text{if } \theta(\mathtt{S}) = S \uplus \{n\} \\ (\textbf{false}, \theta) & \text{otherwise} \end{cases}$$

$$\mathtt{CTN}(n)(\theta) \overset{\text{def}}{=} \begin{cases} (\textbf{true}, \theta) & \text{if } n \in \theta(\mathtt{S}) \\ (\textbf{false}, \theta) & \text{otherwise} \end{cases}$$

(b) Abstract Operations

Figure 40. Lock-Coupling List-Based Set

$$I \overset{\text{def}}{=} \exists A.\ \mathsf{ls}(\mathtt{Head}, A, \mathtt{null}) * \mathsf{s}(A)$$

$$\mathsf{N}_s(x,v,y) \overset{\text{def}}{=} x \mapsto (s,v,y) \qquad \mathsf{N}(x,v,y) \overset{\text{def}}{=} \mathsf{N}_{\_}(x,v,y) \qquad \mathsf{U}(x,v,y) \overset{\text{def}}{=} \mathsf{N}_0(x,v,y) \qquad \mathsf{L}_t(x,v,y) \overset{\text{def}}{=} \mathsf{N}_0(x,v,y) \wedge t > 0$$

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

$$\mathsf{sorted}(A) \overset{\text{def}}{=} \begin{cases} \textbf{true} & \text{if } A = \epsilon \vee A = v :: \epsilon \\ (v_1 < v_2) \wedge \mathsf{sorted}(v_2 :: A') & \text{if } A = v_1 :: v_2 :: A' \end{cases}$$

$$\mathsf{elems}(A) \overset{\text{def}}{=} \begin{cases} \emptyset & \text{if } A = \epsilon \\ \{v\} \cup \mathsf{elems}(A') & \text{if } A = v :: A' \end{cases}$$

$$\mathsf{s}(A) \overset{\text{def}}{=} \exists B.\ (A = \mathtt{MIN} :: B :: \mathtt{MAX}) * (\mathtt{S} \Mapsto \mathsf{elems}(B)) \wedge \mathsf{sorted}(A)$$

$$R_{\mathsf{t}} \overset{\text{def}}{=} \bigvee_{\mathsf{t}' \neq \mathsf{t}} G_{\mathsf{t}'}$$

$$G_{\mathsf{t}} \overset{\text{def}}{=} [\mathsf{Add}_{\mathsf{t}} \vee \mathsf{Rmv}_{\mathsf{t}} \vee \mathsf{Lock}_{\mathsf{t}} \vee \mathsf{Unlock}_{\mathsf{t}}]_I$$

$$\mathsf{Add}_{\mathsf{t}} \overset{\text{def}}{=} \exists x,y,z,n,u,v,w,S.\ (\mathsf{L}_{\mathsf{t}}(x,u,z) * \mathsf{N}(z,w,n) * (\mathtt{S} \Mapsto S) \wedge (u < v < w)) \ltimes (\mathsf{L}_{\mathsf{t}}(x,u,y) * \mathsf{U}(y,v,z) * \mathsf{N}(z,w,n) * (\mathtt{S} \Mapsto S \cup \{v\}))$$

$$\mathsf{Rmv}_{\mathsf{t}} \overset{\text{def}}{=} \exists x,y,z,u,v,S.\ (\mathsf{L}_{\mathsf{t}}(x,u,y) * \mathsf{L}_{\mathsf{t}}(y,v,z) * (\mathtt{S} \Mapsto S) \wedge (v < \mathtt{MAX})) \ltimes (\mathsf{L}_{\mathsf{t}}(x,u,z) * (\mathtt{S} \Mapsto S \backslash \{v\}))$$

$$\mathsf{Lock}_{\mathsf{t}} \overset{\text{def}}{=} \exists x,v,y.\ \mathsf{U}(x,v,y) \ltimes \mathsf{L}_{\mathsf{t}}(x,v,y) \qquad \mathsf{Unlock}_{\mathsf{t}} \overset{\text{def}}{=} \exists x,v,y.\ \mathsf{L}_{\mathsf{t}}(x,v,y) \ltimes \mathsf{U}(x,v,y)$$

$$\mathsf{adjacent}(p,v,c,u) \overset{\text{def}}{=} \exists A,B,z.\ \mathsf{ls}(\mathtt{Head},A,p) * \mathsf{L}_{\mathtt{cid}}(p,v,c) * \mathsf{N}(c,u,z) * \mathsf{ls}(z,B,\mathtt{null}) * \mathsf{s}(A :: v :: u :: B)$$

$$\mathsf{adjacentLocked}(x,v,y,u,z) \overset{\text{def}}{=} \exists A,B.\ \mathsf{ls}(\mathtt{Head},A,p) * \mathsf{L}_{\mathtt{cid}}(x,v,y) * \mathsf{L}_{\mathtt{cid}}(y,u,z) * \mathsf{ls}(z,B,\mathtt{null}) * \mathsf{s}(A :: v :: u :: B)$$

Figure 41. Precise Invariant, Rely and Guarantee of Lock-Coupling List (for Thread t)

E.7 Optimistic List

Next, we verify the optimistic list in Herlihy and Shavit's book [15].

As shown in Figure 45, this implementation traverses the list without taking any locks, and when finding the candidate nodes, it locks the nodes and validates that they are still in the list and adjacent. If the validation fails, the nodes are unlocked and the operation is restarted.

The linearization points are the same as in the lock-coupling list algorithm, where we insert **linself** as usual. We define the precise invariant, the rely and the guarantee in Figure 46, and show the proofs in Figures 47 and 48.

As for the lock-coupling list, the invariant I defined in Figure 46 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. The precise invariant I should include those nodes (garb).

The guarantee G in Figure 46 still contains the Lock, Unlock, Add and Rmv actions. Their definitions are almost the same as in the lock-coupling list (Figure 41), except that after the Rmv action, the removed node is still shared and we just add it to GN.

We give the proofs for the `rmv` and `ctn` methods in Figures 47 and 48 respectively. The proof for the `add` method is similar. The tricky and the most important part is to verify the `validate` function (the proof is shown in Figure 48). This function takes two locked nodes, and re-traverses the list and checks whether they are still in the list and adjacent. But in this traversal, it may access the nodes which have been removed by a concurrent `rmv` method. This does not matter, because:

1. In the algorithm, once a node has been unlinked from the list, the value of its `next` field does not change, thus following the links from the removed node eventually leads back to the list.
2. Any removed node encountered in this traversal must be unlinked from the list *after* the `validate` method started. Thus these removed nodes should be disjoint from both the list and the two locked nodes for validation, even if the two locked nodes have been removed.

Based on the above two observations, we can have the following assertion in the loop invariant in the `validate` method:

$$\mathsf{L_t}(\mathtt{p}, u, \_) * \mathsf{L_t}(\mathtt{c}, v, \_) * \mathsf{ls}(\mathtt{Head}, \_, x) * \mathsf{ls}(\mathtt{s}, \_, x) * \mathsf{true}$$

where `p` and `c` are the two locked nodes for validation, and `s` is the current node in the traversal. The `validate` function first checks whether `p` equals `c`. If so, then we know `s` must be equal to x in the above assertion. Thus `p` must be on the list. If `p` is also the predecessor of `c`, `validate` returns **true**.

```
add(e) :

  local p, c, n;
  while (true) {
    p := Head;
    c := p.next;
    while (c.data < e) {
      p := c;
      c := c.next;
    }
    lock(p);
    lock(c);
    if (validate(p, c)) {
      if (c.data != e) {
        n := cons(0, e, c);
        p.next := n;
        unlock(p);
        unlock(c);
        return true;
      }
      else {
        unlock(p);
        unlock(c);
        return false;
      }
    }
    unlock(p);
    unlock(c);
  }
```

```
rmv(e) :

  local p, c, n;
  while (true) {
    p := Head;
    c := p.next;
    while (c.data < e) {
      p := c;
      c := c.next;
    }
    lock(p);
    lock(c);
    if (validate(p, c)) {
      if (c.data = e) {
        n := c.next;
        p.next := n;
        unlock(p);
        unlock(c);
        return true;
      }
      else {
        unlock(p);
        unlock(c);
        return false;
      }
    }
    unlock(p);
    unlock(c);
  }
```

```
ctn(e) :
  local p, c, n;
  while (true) {
    p := Head;
    c := p.next;
    while (c.data < e) {
      p := c;
      c := c.next;
    }
    lock(p);
    lock(c);
    if (validate(p, c)) {
      unlock(p);
      unlock(c);
      return (c.data = e);
    }
    unlock(p);
    unlock(c);
  }

validate(p, c):
  local s;
  s := Head;
  while (s.data <= p.data) {
    if (s = p)
      return (p.next = c);
    s := s.next;
  }
  return false;
```

Implementation (from Herlihy & Shavit's book)

Figure 45. Optimistic List

$I \overset{\text{def}}{=} \exists A.\ \mathsf{ls}(\mathtt{Head}, A, \mathtt{null}) * \mathsf{s}(A) * \mathsf{garb}$

$\mathsf{N}_s(x,v,y) \overset{\text{def}}{=} x \mapsto (s,v,y) \qquad \mathsf{N}(x,v,y) \overset{\text{def}}{=} \mathsf{N}_\_(x,v,y) \qquad \mathsf{U}(x,v,y) \overset{\text{def}}{=} \mathsf{N}_0(x,v,y) \qquad \mathsf{L}_t(x,v,y) \overset{\text{def}}{=} \mathsf{N}_0(x,v,y) \wedge t > 0$

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

$$\mathsf{sorted}(A) \overset{\text{def}}{=} \begin{cases} \textbf{true} & \text{if } A = \epsilon \vee A = v :: \epsilon \\ (v_1 < v_2) \wedge \mathsf{sorted}(v_2 :: A') & \text{if } A = v_1 :: v_2 :: A' \end{cases}$$

$$\mathsf{elems}(A) \overset{\text{def}}{=} \begin{cases} \emptyset & \text{if } A = \epsilon \\ \{v\} \cup \mathsf{elems}(A') & \text{if } A = v :: A' \end{cases}$$

$\mathsf{s}(A) \overset{\text{def}}{=} \exists B.\ (A = \mathtt{MIN} :: B :: \mathtt{MAX}) * (\mathtt{S} \Mapsto \mathsf{elems}(B)) \wedge \mathsf{sorted}(A) \qquad \mathsf{garb} \overset{\text{def}}{=} \circledast_{x \in \mathtt{GN}}.\mathsf{N}(x, \_, \_)$

$R_{\mathsf{t}} \overset{\text{def}}{=} \bigvee_{\mathsf{t}' \neq \mathsf{t}} G_{\mathsf{t}'}$

$G_{\mathsf{t}} \overset{\text{def}}{=} [\mathsf{Add}_{\mathsf{t}} \vee \mathsf{Rmv}_{\mathsf{t}} \vee \mathsf{Lock}_{\mathsf{t}} \vee \mathsf{Unlock}_{\mathsf{t}}]_I$

$\mathsf{Add}_{\mathsf{t}} \overset{\text{def}}{=} \exists x,y,z,n,u,v,w,S.\ (\mathsf{L}_{\mathsf{t}}(x,u,z) * \mathsf{N}(z,w,n) * (\mathtt{S} \Mapsto S) \wedge (u < v < w)) \ltimes (\mathsf{L}_{\mathsf{t}}(x,u,y) * \mathsf{U}(y,v,z) * \mathsf{N}(z,w,n) * (\mathtt{S} \Mapsto S \cup \{v\}))$

$\mathsf{Rmv}_{\mathsf{t}} \overset{\text{def}}{=} \exists x,y,z,u,v,S_g,S.\ (\mathsf{L}_{\mathsf{t}}(x,u,y) * \mathsf{L}_{\mathsf{t}}(y,v,z) * (\mathtt{GN} = S_g) * (\mathtt{S} \Mapsto S) \wedge (v < \mathtt{MAX}))$
$\quad \ltimes (\mathsf{L}_{\mathsf{t}}(x,u,z) * \mathsf{L}_{\mathsf{t}}(y,v,z) * (\mathtt{GN} = S_g \cup \{y\}) * (\mathtt{S} \Mapsto S \backslash \{v\}))$

$\mathsf{Lock}_{\mathsf{t}} \overset{\text{def}}{=} \exists x,v,y.\ \mathsf{U}(x,v,y) \ltimes \mathsf{L}_{\mathsf{t}}(x,v,y) \qquad \mathsf{Unlock}_{\mathsf{t}} \overset{\text{def}}{=} \exists x,v,y.\ \mathsf{L}_{\mathsf{t}}(x,v,y) \ltimes \mathsf{U}(x,v,y)$

Figure 46. Precise Invariant, Rely and Guarantee of Optimistic List (for Thread t)

```
IntSet GN;   //Auxiliary global variable for verification: removed nodes

rmv(e):
  local p, c, n;
 $\{I * \mathsf{t} \rightarrowtail (\mathtt{RMV}, \mathsf{e}) \wedge (\mathtt{MIN} < \mathsf{e} < \mathtt{MAX})\}$
  while (true) {
    p := Head;
    c := p.next;
    while (c.data < e) {
      p := c;
      c := c.next;
    }
    lock(p);
    lock(c);
   $\{\exists u, v.\ (I \wedge \mathsf{L_t}(\mathsf{p}, u, \_) * \mathsf{L_t}(\mathsf{c}, v, \_) * \mathsf{true}) * \mathsf{t} \rightarrowtail (\mathtt{RMV}, \mathsf{e}) \wedge (u < \mathsf{e} \le v)\}$
    if (validate(p, c)) {
     $\{\exists u, v.\ (I \wedge \mathsf{ls}(\mathtt{Head}, \_, \mathsf{p}) * \mathsf{L_t}(\mathsf{p}, u, \mathsf{c}) * \mathsf{L_t}(\mathsf{c}, v, \_) * \mathsf{true}) * \mathsf{t} \rightarrowtail (\mathtt{RMV}, \mathsf{e}) \wedge (u < \mathsf{e} \le v)\}$
      if (c.data = e) {
        n := c.next;
       $\{\exists u.\ (I \wedge \mathsf{ls}(\mathtt{Head}, \_, \mathsf{p}) * \mathsf{L_t}(\mathsf{p}, u, \mathsf{c}) * \mathsf{L_t}(\mathsf{c}, \mathsf{e}, \mathsf{n}) * \mathsf{true}) * \mathsf{t} \rightarrowtail (\mathtt{RMV}, \mathsf{e})\}$
        < p.next := n; GN := GN ∪ {c}; linself; >
       $\{\exists u.\ (I \wedge \mathsf{ls}(\mathtt{Head}, \_, \mathsf{p}) * \mathsf{L_t}(\mathsf{p}, u, \mathsf{n}) * \mathsf{L_t}(\mathsf{c}, \mathsf{e}, \mathsf{n}) * \mathsf{true}) * \mathsf{t} \rightarrowtail (\mathbf{end}, \mathbf{true})\}$
        unlock(p);
        unlock(c);
       $\{I * \mathsf{t} \rightarrowtail (\mathbf{end}, \mathbf{true})\}$
        return true;
      }
      else {
       $\{\exists u, v.\ (I \wedge \mathsf{ls}(\mathtt{Head}, \_, \mathsf{p}) * \mathsf{L_t}(\mathsf{p}, u, \mathsf{c}) * \mathsf{L_t}(\mathsf{c}, v, \_) * \mathsf{true}) * \mathsf{t} \rightarrowtail (\mathtt{RMV}, \mathsf{e}) \wedge (u < \mathsf{e} < v)\}$
        linself;
       $\{\exists u, v.\ (I \wedge \mathsf{ls}(\mathtt{Head}, \_, \mathsf{p}) * \mathsf{L_t}(\mathsf{p}, u, \mathsf{c}) * \mathsf{L_t}(\mathsf{c}, v, \_) * \mathsf{true}) * \mathsf{t} \rightarrowtail (\mathbf{end}, \mathbf{false}) \wedge (u < \mathsf{e} < v)\}$
        unlock(p);
        unlock(c);
       $\{I * \mathsf{t} \rightarrowtail (\mathbf{end}, \mathbf{false})\}$
        return false;
      }
    }
   $\{\exists u, v.\ (I \wedge \mathsf{L_t}(\mathsf{p}, u, \_) * \mathsf{L_t}(\mathsf{c}, v, \_) * \mathsf{true}) * \mathsf{t} \rightarrowtail (\mathtt{RMV}, \mathsf{e}) \wedge (u < \mathsf{e} \le v)\}$
    unlock(p);
    unlock(c);
  }
```

Figure 47. Proof Outline of Remove of Optimistic List for Thread t

```
ctn(e):
  local p, c, n;
 $\{I * \mathsf{t} \rightarrowtail (\mathtt{CTN}, \mathsf{e}) \wedge (\mathtt{MIN} < \mathsf{e} < \mathtt{MAX})\}$
  while (true) {
    p := Head;
    c := p.next;
   $\{\exists u, x.\ (I \wedge \mathsf{N}(\mathsf{p}, u, \_) * \mathsf{ls}(\mathsf{c}, \_, x) * \mathsf{N}(x, \mathtt{MAX}, \mathtt{null}) * \mathsf{true}) * \mathsf{t} \rightarrowtail (\mathtt{CTN}, \mathsf{e}) \wedge (u < \mathsf{e} < \mathtt{MAX})\}$
    while (c.data < e) {
     $\{\exists u, v, x, y.\ (I \wedge \mathsf{N}(\mathsf{p}, u, \_) * \mathsf{N}(\mathsf{c}, v, x) * \mathsf{ls}(x, \_, y) * \mathsf{N}(y, \mathtt{MAX}, \mathtt{null}) * \mathsf{true}) * \mathsf{t} \rightarrowtail (\mathtt{CTN}, \mathsf{e}) \wedge (v < \mathsf{e} < \mathtt{MAX})\}$
      p := c;
      c := c.next;
    }
   $\{\exists u, v.\ (I \wedge \mathsf{N}(\mathsf{p}, u, \_) * \mathsf{N}(\mathsf{c}, v, \_) * \mathsf{true}) * \mathsf{t} \rightarrowtail (\mathtt{CTN}, \mathsf{e}) \wedge (u < \mathsf{e} \leq v)\}$
    lock(p);
    lock(c);
   $\{\exists u, v.\ (I \wedge \mathsf{L_t}(\mathsf{p}, u, \_) * \mathsf{L_t}(\mathsf{c}, v, \_) * \mathsf{true}) * \mathsf{t} \rightarrowtail (\mathtt{CTN}, \mathsf{e}) \wedge (u < \mathsf{e} \leq v)\}$
    if (validate(p, c)) {
     $\{\exists u, v.\ (I \wedge \mathsf{ls}(\mathtt{Head}, \_, \mathsf{p}) * \mathsf{L_t}(\mathsf{p}, u, \mathsf{c}) * \mathsf{L_t}(\mathsf{c}, v, \_) * \mathsf{true}) * \mathsf{t} \rightarrowtail (\mathtt{CTN}, \mathsf{e}) \wedge (u < \mathsf{e} \leq v)\}$
      linself;
     $\{\exists u, v.\ (I \wedge \mathsf{ls}(\mathtt{Head}, \_, \mathsf{p}) * \mathsf{L_t}(\mathsf{p}, u, \mathsf{c}) * \mathsf{L_t}(\mathsf{c}, v, \_) * \mathsf{true}) * ((\mathsf{e} = v \wedge \mathsf{t} \rightarrowtail (\mathbf{end}, \mathbf{true})) \vee (\mathsf{e} \neq v \wedge \mathsf{t} \rightarrowtail (\mathbf{end}, \mathbf{false}))) \wedge (u < \mathsf{e} \leq v)\}$
      unlock(p);
      unlock(c);
     $\{\exists v.\ (I \wedge \mathsf{N}(\mathsf{c}, v, \_) * \mathsf{true}) * ((\mathsf{e} = v \wedge \mathsf{t} \rightarrowtail (\mathbf{end}, \mathbf{true})) \vee (\mathsf{e} \neq v \wedge \mathsf{t} \rightarrowtail (\mathbf{end}, \mathbf{false})))\}$
      return (c.data = e);
    }
   $\{\exists u, v.\ (I \wedge \mathsf{L_t}(\mathsf{p}, u, \_) * \mathsf{L_t}(\mathsf{c}, v, \_) * \mathsf{true}) * \mathsf{t} \rightarrowtail (\mathtt{CTN}, \mathsf{e}) \wedge (u < \mathsf{e} \leq v)\}$
    unlock(p);
    unlock(c);
  }

validate(p, c):
  local s;
 $\{I \wedge \mathsf{L_t}(\mathsf{p}, u, \_) * \mathsf{L_t}(\mathsf{c}, v, \_) * \mathsf{true} \wedge u < v\}$
  s := Head;
 $\{I \wedge \exists w, x.\ \mathsf{L_t}(\mathsf{p}, u, \_) * \mathsf{L_t}(\mathsf{c}, v, \_) * \mathsf{ls}(\mathtt{Head}, \_, x) * \mathsf{ls}(\mathsf{s}, \_, x) * \mathsf{true} \wedge \mathsf{N}(\mathsf{s}, w, \_) * \mathsf{true} \wedge u < v\}$
  while (s.data <= p.data) {
   $\{I \wedge \exists w, x, y.\ \mathsf{L_t}(\mathsf{p}, u, \_) * \mathsf{L_t}(\mathsf{c}, v, \_) * \mathsf{ls}(\mathtt{Head}, \_, x) * \mathsf{ls}(\mathsf{s}, \_, x) * \mathsf{true} \wedge \mathsf{N}(\mathsf{s}, w, y) * \mathsf{N}(y, \_, \_) * \mathsf{true} \wedge w \leq u < v\}$
    if (s = p)
     $\{I \wedge \mathsf{ls}(\mathtt{Head}, \_, \mathsf{p}) * \mathsf{L_t}(\mathsf{p}, u, \_) * \mathsf{L_t}(\mathsf{c}, v, \_) * \mathsf{true} \wedge u < v\}$
      return (p.next = c);
   $\left\{ \begin{array}{l} \ \ I \wedge \exists w, y.\ \mathsf{L_t}(\mathsf{p}, u, \_) * \mathsf{L_t}(\mathsf{c}, v, \_) * \mathsf{ls}(\mathtt{Head}, \_, \mathsf{s}) * \mathsf{N}(\mathsf{s}, w, y) * \mathsf{true} \wedge \mathsf{N}(y, \_, \_) * \mathsf{true} \wedge w \leq u < v \\ \vee\ I \wedge \exists w, x, y.\ \mathsf{L_t}(\mathsf{p}, u, \_) * \mathsf{L_t}(\mathsf{c}, v, \_) * \mathsf{ls}(\mathtt{Head}, \_, x) * \mathsf{N}(\mathsf{s}, w, y) * \mathsf{ls}(y, \_, x) * \mathsf{true} \wedge \mathsf{N}(y, \_, \_) * \mathsf{true} \wedge w \leq u < v \end{array} \right\}$
    s := s.next;
  }
 $\{I \wedge \mathsf{L_t}(\mathsf{p}, u, \_) * \mathsf{L_t}(\mathsf{c}, v, \_) * \mathsf{true} \wedge u < v\}$
  return false;
```

Figure 48. Proof Outline of Contains and Validate of Optimistic List for Thread t

E.8 Lazy List

The lazy list algorithm [13] has a wait-free `ctn` method. As shown in Figure 49, every node in the concrete list has a `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). The `ctn(e)` method traverses the list once ignoring the locks on the nodes, and returns **true** if it can find an unmarked `e`, and **false** otherwise.

Linearization points. The linearization points of `add` and `rmv` can be statically located in the method code. Just note that a successful `rmv` is linearized when the node is *logically* removed.

A successful `ctn` is linearized when an unmarked matching node is found. However, the LP of an unsuccessful `ctn` might depend on the future interleavings with the sibling threads. Following Vafeiadis [31], we can linearize the read-only `ctn(e)` method multiple times according to the following principles:

1. At the beginning of its execution, we linearize the method if `e` is *not* in the list.
2. Whenever some sibling thread removes `e` from the list, we linearize the pending `ctn(e)` method if it has not reached the linearization point for successful searching (in 3).
3. When `e` is successfully found, we linearize `ctn(e)`.

The first two principles include all the scenarios when `ctn(e)` returns **false**, which ensure that at any time in its executions, if the method has not been linearized, then `e` must be in the set.

To help specify the linearization points for `ctn`, we use a global auxiliary variable `OutOps`, a set containing the information of all the pending `ctn` operations. We introduce auxiliary code in the `ctn` and `rmv` methods, as shown by the red-colored code in Figures 51 and 52. At the beginning of `ctn`, we allocate the thread record and add it to `OutOps`. A thread record contains the thread identifier `t`, the argument `e`, and a field `res` to record the current status (which is `UNDEF` initially, and **false** when the method has been linearized). Then according to the above three principles, we set the `res` field to **false** at the beginning of the `ctn` if `e` is not in the list. Also `rmv(e)` will set the field at the time when `e` is logically removed. We will delete the thread record from `OutOps` at the last possible linearization point of `ctn`, *i.e.*, the LP for successful searching.

Then, as shown in the highlighted code in Figures 51 and 52, we insert **trylinself** and **trylin** at potential LPs in the `ctn` and `rmv` methods, and **commit** at the time when we know the return value of the `ctn` method. Note we can insert **linself** at the LP for successful `ctn`s, since this is a LP for sure.

Although we introduce the global auxiliary variable `OutOps`, our verification is still thread-local. We treat `OutOps` as a normal shared variable, and specify it in pre- and post-conditions and rely/guarantee conditions. We do not need to know the number of threads and which method is invoked by which thread.

Invariant, rely and guarantee. We define the precise invariant, the rely and the guarantee in Figure 50. As in the optimistic list, the invariant I contains the concrete list and the abstract set, and also the removed nodes (garb) specified by the auxiliary variable `GN`. The values of *unmarked* nodes on the list constitute the abstract set `S`. In addition, I also contains the thread descriptors in `OutOps` and the corresponding abstract operations. The `res` field of a thread descriptor tells us the abstract operation of that thread, as defined in $\mathsf{D}(d, t, n)$. If `res` of the descriptor d is `UNDEF`, then the thread t must have not done its $(\mathtt{CTN}, n)$ operation; otherwise, the thread has passed its potential LP and we can guess its abstract operation is the original $(\mathtt{CTN}, n)$ or (**end**, **false**).

The atomic actions of the algorithm include locking a node (Lock), releasing a lock (Unlock), adding a node (Add), physically removing a marked node (Rmv), marking a node of value v and setting the `res` fields of the thread records in `OutOps` for the threads who are searching for v (Mark), adding the record of the current thread to `OutOps` (AddOut) and removing its record (RmvOut). Note that when defining Mark, we can use $*$ to separate the actions on the `mark` field of the node and on the `res` fields of descriptors, which are simultaneous. The latter is specified by the action TrylinOut. All these actions form the guarantee G of a thread, and the rely R is the union of the actions made by all the other threads.

Proofs. We show the verification of the `ctn` and `rmv` methods in Figures 51 and 52. The proofs are straightforward, following our inference rules.

```
struct Node{ int lock; int val; Node *next; bool mark; };
```

```
locate(e) :
    local p, c;
    while (true) {
      p := Head;
      c := p.next;
      while (c.val < e) {
        p := c;
        c := c.next;
      }
      lock(p);
      lock(c);
      if (!p.mark && !c.mark
           && p.next = c)
        return (p, c);
      else {
        unlock(p);
        unlock(c);
      }
    }
```

```
add(e) :
    local p, c, n, r;
    (p, c) := locate(e);
    if (c.val != e) {
      n := cons(0, e, c, false);
      p.next := n;
      r := true;
    }
    else {
      r := false;
    }
    unlock(p);
    unlock(c);
    return r;
```

```
rmv(e) :
    local p, c, n, r;
    (p, c) := locate(e);
    if (c.val = e) {
      c.mark := true;
      n := c.next;
      p.next := n;
      r := true;
    }
    else {
      r := false;
    }
    unlock(p);
    unlock(c);
    return r;
```

```
ctn(e) :
    local c;
    c := Head;
    while (c.val < e) {
      c := c.next;
    }
    b := c.mark;
    if (!b && c.val = e)
      return true;
    else
      return false;
```

Figure 49. Lazy List

$$I \overset{\text{def}}{=} \exists A.\ \mathsf{ls}(\mathtt{Head}, A, \mathtt{null}) * \mathsf{s}(A) * \mathsf{Ds}(\mathtt{OutOps}) * \mathsf{garb}$$

$$\mathsf{N}(x,v,y,b) \overset{\text{def}}{=} x \mapsto (\_, v, y, b) \qquad \mathsf{L}_{\mathsf{t}}(x,v,y,b) \overset{\text{def}}{=} x \mapsto (\mathsf{t}, v, y, b) \wedge (\mathsf{t} > 0) \qquad \mathsf{U}(x,v,y,b) \overset{\text{def}}{=} x \mapsto (0, v, y, b)$$

$$\mathsf{ls}(x,A,z) \overset{\text{def}}{=} (x = z \wedge A = \epsilon) \vee (x \neq z \wedge \exists v, y, b, A'.\ \mathsf{N}(x,v,y,b) * \mathsf{ls}(y, A', z) \wedge A = (v,b) :: A')$$

$$\mathsf{sorted}(A) \overset{\text{def}}{=} \begin{cases} \textbf{true} & \text{if } A = \epsilon \vee A = (v,b) :: \epsilon \\ (v_1 < v_2) \wedge \mathsf{sorted}((v_2,b_2) :: A') & \text{if } A = (v_1,b_1) :: (v_2,b_2) :: A' \end{cases}$$

$$\mathsf{elems}(A) \overset{\text{def}}{=} \begin{cases} \emptyset & \text{if } A = \epsilon \\ \{v\} \cup \mathsf{elems}(A') & \text{if } A = (v, \textbf{false}) :: A' \\ \mathsf{elems}(A') & \text{if } A = (v, \textbf{true}) :: A' \end{cases}$$

$$\mathsf{s}(A) \overset{\text{def}}{=} \exists A'.\ (A = (\mathtt{MIN}, \textbf{false}) :: A' :: (\mathtt{MAX}, \textbf{false})) \wedge \mathsf{sorted}(A) \wedge (\mathtt{S} \Mapsto \mathsf{elems}(A')) \qquad \mathsf{garb} \overset{\text{def}}{=} \circledast_{x \in \mathtt{GN}}.\mathsf{N}(x, \_, \_, \_)$$

$$\mathsf{d}(d,t,n,b) \overset{\text{def}}{=} d \mapsto (t,n,b) \wedge (b = \mathtt{UNDEF} \vee b = \textbf{false})$$

$$\mathsf{notDone}(d,t,n) \overset{\text{def}}{=} \mathsf{d}(d,t,n,\mathtt{UNDEF}) * t \rightarrowtail (\mathtt{CTN}, n)$$

$$\mathsf{afterTrylin}(d,t,n) \overset{\text{def}}{=} \mathsf{d}(d,t,n,\textbf{false}) * (t \rightarrowtail (\mathtt{CTN}, n) \oplus t \rightarrowtail (\textbf{end}, \textbf{false}))$$

$$\mathsf{D}(d,t,n) \overset{\text{def}}{=} \mathsf{notDone}(d,t,n) \vee \mathsf{afterTrylin}(d,t,n) \qquad \mathsf{Ds}(O) \overset{\text{def}}{=} \circledast_{d \in O}.\mathsf{D}(d, \_, \_)$$

$$R_{\mathsf{t}} \overset{\text{def}}{=} \bigvee_{\mathsf{t}' \neq \mathsf{t}} G_{\mathsf{t}'}$$

$$G_{\mathsf{t}} \overset{\text{def}}{=} [\mathsf{Lock}_{\mathsf{t}} \vee \mathsf{Unlock}_{\mathsf{t}} \vee \mathsf{Add}_{\mathsf{t}} \vee \mathsf{Mark}_{\mathsf{t}} \vee \mathsf{Rmv}_{\mathsf{t}} \vee \mathsf{AddOut}_{\mathsf{t}} \vee \mathsf{RmvOut}_{\mathsf{t}}]_I$$

$$\mathsf{Lock}_{\mathsf{t}} \overset{\text{def}}{=} \exists x, v, y, b.\ \mathsf{U}(x,v,y,b) \ltimes \mathsf{L}_{\mathsf{t}}(x,v,y,b) \qquad \mathsf{Unlock}(\mathsf{t}) \overset{\text{def}}{=} \exists x, v, y, b.\ \mathsf{L}_{\mathsf{t}}(x,v,y,b) \ltimes \mathsf{U}(x,v,y,b)$$

$$\begin{aligned} \mathsf{Add}_{\mathsf{t}} \overset{\text{def}}{=} {} & \exists x, y, z, p, u, v, w, S.\ (\mathsf{L}_{\mathsf{t}}(x,u,y,\textbf{false}) * \mathsf{L}_{\mathsf{t}}(y,w,z,\textbf{false}) * (\mathtt{S} \Mapsto S) \wedge (u < v < w)) \\ & \ltimes (\mathsf{L}_{\mathsf{t}}(x,u,p,\textbf{false}) * \mathsf{U}(p,v,y,\textbf{false}) * \mathsf{L}_{\mathsf{t}}(y,w,z,\textbf{false}) * (\mathtt{S} \Mapsto S \cup \{v\})) \end{aligned}$$

$$\mathsf{Mark}_{\mathsf{t}} \overset{\text{def}}{=} \exists y, z, v, S, O.\ ((\mathsf{L}_{\mathsf{t}}(y,v,z,\textbf{false}) * (\mathtt{S} \Mapsto S \cup \{v\}) \wedge (v < \mathtt{MAX})) \ltimes (\mathsf{L}_{\mathsf{t}}(y,v,z,\textbf{true}) * (\mathtt{S} \Mapsto S))) * \mathsf{TrylinOut}(v, O)$$

$$\mathsf{TrylinOut}(O,v) \overset{\text{def}}{=} \circledast_{d \in O}.(\exists t.\ \mathsf{D}(d,t,v) \ltimes \mathsf{afterTrylin}(d,t,v)) \qquad \text{provided } \exists O'.\ (\mathtt{OutOps} = O \uplus O') \wedge \forall d \in O'.\ \exists n.\ (n \neq v) \wedge \mathsf{D}(d, \_, n)$$

$$\mathsf{Rmv}_{\mathsf{t}} \overset{\text{def}}{=} \exists x, y, z, u, v, S_g.\ (\mathsf{L}_{\mathsf{t}}(x,u,y,\textbf{false}) * \mathsf{L}_{\mathsf{t}}(y,v,z,\textbf{true}) * (\mathtt{GN} = S_g) \wedge (v < \mathtt{MAX})) \ltimes (\mathsf{L}_{\mathsf{t}}(x,u,z,\textbf{false}) * \mathsf{L}_{\mathsf{t}}(y,v,z,\textbf{true}) * (\mathtt{GN} = S_g \cup \{y\}))$$

$$\mathsf{AddOut}_{\mathsf{t}} \overset{\text{def}}{=} \exists O, d.\ (\mathtt{OutOps} = O) \ltimes ((\mathtt{OutOps} = O \uplus \{d\}) * \mathsf{D}(d, \mathsf{t}, \_))$$

$$\mathsf{RmvOut}_{\mathsf{t}} \overset{\text{def}}{=} \exists O, d.\ ((\mathtt{OutOps} = O \uplus \{d\}) * \mathsf{D}(d, \mathsf{t}, \_)) \ltimes (\mathtt{OutOps} = O)$$

Figure 50. Precise Invariant, Rely and Guarantee of Lazy List (for Thread t)

Always $*$garb

`IntSet OutOps;` //Auxiliary global variable for verification: thread descriptors for contains

```
ctn(e):
local c, b, d, ac;
$\{\exists A.\ \mathsf{ls}(\mathtt{Head}, A, \mathtt{null}) * \mathsf{s}(A) * \mathsf{Ds}(\mathtt{OutOps}) * \mathsf{t} \rightarrowtail (\mathtt{CTN}, \mathsf{e})\}$
< d := cons(t, e, UNDEF);
  ac := Head; while(ac.val < e) ac := ac.next;
  if (ac.mark || ac.val!=e) { d.res:= false; trylinself; }
  OutOps := OutOps∪{d}; >
$\{\exists O.\ \mathtt{OutOps} = O \uplus \{\mathsf{d}\}$
 $\wedge((\exists A, B, x, y.\ \mathsf{ls}(\mathtt{Head}, A, x) * \mathsf{N}(x, \mathsf{e}, y, \mathbf{false}) * \mathsf{ls}(y, B, \mathtt{null})$
   $* \mathsf{Ds}(O) * \mathsf{notDone}(\mathsf{d}, \mathsf{t}, \mathsf{e}) * \mathsf{s}(A :: (\mathsf{e}, \mathbf{false}) :: B))$
 $\vee(\exists A.\mathsf{ls}(\mathtt{Head}, A, \mathtt{null}) * \mathsf{Ds}(O) * \mathsf{afterTrylin}(\mathsf{d}, \mathsf{t}, \mathsf{e}) * \mathsf{s}(A)))\}$
c := Head;
$\{\exists O.\ \mathtt{OutOps} = O \uplus \{\mathsf{d}\}$
 $\wedge((\exists A, A', B, x, y.\ \mathsf{ls}(\mathtt{Head}, A, \mathsf{c}) * \mathsf{ls}(\mathsf{c}, A', x) * \mathsf{N}(x, \mathsf{e}, y, \mathbf{false})$
   $* \mathsf{ls}(y, B, \mathtt{null}) * \mathsf{Ds}(O) * \mathsf{notDone}(\mathsf{d}, \mathsf{t}, \mathsf{e}) * \mathsf{s}(A :: A' :: (\mathsf{e}, \mathbf{false}) :: B))$
 $\vee(\exists A, B, y, v, b.\ \mathsf{ls}(\mathtt{Head}, A, \mathsf{c}) * \mathsf{N}(\mathsf{c}, v, y, b) * \mathsf{ls}(y, B, \mathtt{null})$
   $* \mathsf{Ds}(O) * \mathsf{afterTrylin}(\mathsf{d}, \mathsf{t}, \mathsf{e}) * \mathsf{s}(A :: (v, b) :: B)))\}$
while (c.val < e) { c := c.next; }
$\{\exists O.\ \mathtt{OutOps} = O \uplus \{\mathsf{d}\}$
 $\wedge((\exists A, B, y.\ \mathsf{ls}(\mathtt{Head}, A, \mathsf{c}) * \mathsf{N}(\mathsf{c}, \mathsf{e}, y, \mathbf{false}) * \mathsf{ls}(y, B, \mathtt{null})$
   $* \mathsf{Ds}(O) * \mathsf{notDone}(\mathsf{d}, \mathsf{t}, \mathsf{e}) * \mathsf{s}(A :: (\mathsf{e}, \mathbf{false}) :: B))$
 $\vee(\exists A, B, y, v, b.\ \mathsf{ls}(\mathtt{Head}, A, \mathsf{c}) * \mathsf{N}(\mathsf{c}, v, y, b) * \mathsf{ls}(y, B, \mathtt{null})$
   $* \mathsf{Ds}(O) * \mathsf{afterTrylin}(\mathsf{d}, \mathsf{t}, \mathsf{e}) * \mathsf{s}(A :: (v, b) :: B) \wedge v \geq \mathsf{e}))\}$
< b := c.mark;
  if (!b && c.val=e) { linself; commit(t ↣ (end,true)); } else { commit(t ↣ (end,false)); }
  OutOps := OutOps\{d}; dispose(d); >
$\{\exists A, B, v, y.\ \mathsf{ls}(\mathtt{Head}, A, \mathsf{c}) * \mathsf{N}(\mathsf{c}, v, y, \mathsf{b}) * \mathsf{ls}(y, B, \mathtt{null})$
 $*((\mathsf{t} \rightarrowtail (\mathbf{end}, \mathbf{true}) \wedge \neg\mathsf{b} \wedge v = \mathsf{e}) \vee (\mathsf{t} \rightarrowtail (\mathbf{end}, \mathbf{false}) \wedge (\mathsf{b} \vee v > \mathsf{e})))$
 $* \mathsf{D}(\mathtt{OutOps}) * \mathsf{s}(A :: (v, b) :: B)\}$
if (!b && c.val=e)
  $\{\exists A.\ \mathsf{ls}(\mathtt{Head}, A, \mathtt{null}) * \mathsf{s}(A) * \mathsf{Ds}(\mathtt{OutOps}) * (\mathsf{t} \rightarrowtail (\mathbf{end}, \mathbf{true}))\}$
  return true;
else
  $\{\exists A.\ \mathsf{ls}(\mathtt{Head}, A, \mathtt{null}) * \mathsf{s}(A) * \mathsf{Ds}(\mathtt{OutOps}) * (\mathsf{t} \rightarrowtail (\mathbf{end}, \mathbf{false}))\}$
  return false;
```

Figure 51. Proof Outline of Contains of Lazy List for Thread t

$$\mathsf{adjacentLocked}(x,v,y,u,z,b,b') \overset{\text{def}}{=} \exists A,B.\ \mathsf{ls}(\mathtt{Head},A,x) * \mathsf{L_t}(x,v,y,b) * \mathsf{L_t}(y,u,z,b') * \mathsf{ls}(z,B,\mathtt{null}) * \mathsf{s}(A::(v,b)::(u,b')::B)$$

```
IntSet GN;    //Auxiliary global variable for verification: removed nodes

rmv(e):
local p, c, n, r, d;
{$I * \mathsf{t} \rightarrowtail (\mathtt{RMV},\mathsf{e}) \wedge (\mathsf{e} < \mathtt{MAX})$}
(p, c) := locate(e);
{$\exists u,v.\ \mathsf{adjacentLocked}(\mathsf{p},u,\mathsf{c},v,\_,\mathbf{false},\mathbf{false}) * \mathsf{Ds}(\mathtt{OutOps}) * \mathsf{t} \rightarrowtail (\mathtt{RMV},\mathsf{e}) * \mathsf{garb} \wedge (u < \mathsf{e} \le v) \wedge (\mathsf{e} < \mathtt{MAX})$}
if (c.val = e) {
  {$\exists u.\ \mathsf{adjacentLocked}(\mathsf{p},u,\mathsf{c},\mathsf{e},\_,\mathbf{false},\mathbf{false}) * \mathsf{Ds}(\mathtt{OutOps}) * \mathsf{t} \rightarrowtail (\mathtt{RMV},\mathsf{e}) * \mathsf{garb} \wedge (\mathsf{e} < \mathtt{MAX})$}
  < c.mark := true;  linself;
    foreach d in OutOps
      if (d.arg = e) { d.res:= false; trylin(d.id); } >
  {$\exists u.\ \mathsf{adjacentLocked}(\mathsf{p},u,\mathsf{c},\mathsf{e},\_,\mathbf{false},\mathbf{true}) * \mathsf{Ds}(\mathtt{OutOps}) * \mathsf{t} \rightarrowtail (\mathbf{end},\mathbf{true}) * \mathsf{garb} \wedge (\mathsf{e} < \mathtt{MAX})$}
  n := c.next;
  < p.next := n; GN := GN $\cup$ {c}; >
  {$\exists A,B,u.\ \mathsf{ls}(\mathtt{Head},A,\mathsf{p}) * \mathsf{L_t}(\mathsf{p},u,\mathsf{n},\mathbf{false}) * \mathsf{ls}(\mathsf{n},B,\mathtt{null}) * \mathsf{Ds}(\mathtt{OutOps})$
   $* \mathsf{t} \rightarrowtail (\mathbf{end},\mathbf{true}) * \mathsf{s}(A::(u,\mathbf{false})::B) * \mathsf{garb}$}
  r := true;
}
else {
  {$\exists u,v.\ \mathsf{adjacentLocked}(\mathsf{p},u,\mathsf{c},v,\_,\mathbf{false},\mathbf{false}) * \mathsf{Ds}(\mathtt{OutOps}) * \mathsf{t} \rightarrowtail (\mathtt{RMV},\mathsf{e}) * \mathsf{garb} \wedge (u < \mathsf{e} < v)$}
  linself;
  {$\exists u,v.\ \mathsf{adjacentLocked}(\mathsf{p},u,\mathsf{c},v,\_,\mathbf{false},\mathbf{false}) * \mathsf{Ds}(\mathtt{OutOps}) * \mathsf{t} \rightarrowtail (\mathbf{end},\mathbf{false}) * \mathsf{garb}$}
  r := false;
}
unlock(p);
unlock(c);
{$I * \mathsf{t} \rightarrowtail (\mathbf{end},\mathsf{r})$}
return r;
```

Figure 52. Proof Outline of Remove of Lazy List for Thread t

E.9 Lock-Free List

The last list-based set algorithm is the classical Harris-Michael list algorithm [11, 21]. We show a variation in Figure 53, which is is taken from Herlihy and Shavit's book [15], and has been corrected according to its errata.

The algorithm does not use locks, but takes full advantage of the `mark` field of the list node. To avoid missing the effects of `add` or `rmv` operations, we need to ensure that a node's field cannot be updated if the node has been logically or physically removed. Thus the algorithm uses a new style of `cas`:

```
cas(&(p.(next, mark)), o, n, b, b')
```

It updates `p.next` and `p.mark` atomically to `n` and `b'`, if they are originally `o` and `b` respectively. The following command allows reading `p.next` and `p.mark` atomically:

```
(n, b) := p.(next, mark);
```

In other words, we treat the node's `next` and `mark` fields as a single atomic unit. Then in the algorithm, we update the `next` field only when the `mark` field is **true**.

An idea of the algorithm is to let every thread doing `add` or `rmv` physically remove all marked nodes it encounters, before doing its own operation. This is a "helping mechanism", although it does not affect the location of linearization points (since the LP of a `rmv` is at the time of marking the node, rather than physically removal). For `ctn`, the version we verified in this paper uses the same code of lazy list, thus differs from Harris' or Michael's original version. In Harris' or Michael's version, the `ctn` method will call the `locate` function, thus also helps physically remove the marked nodes. Both versions have benefits and drawbacks in different situations. For verification, the difference is reflected in the location of the linearization point for unsuccessful contains. In Harris' or Michael's version, its LPs can be located in the thread currently being verified; while in Herlihy and Shavit's version we verified, the LP might be in other threads and depend on future, just like the lazy list.

We define the precise invariant, the rely and the guarantee in Figure 54. The definitions are similar to those for the lazy list 50. We only want to emphasize the definitions for the Add, Mark and Rmv actions, which all require the predecessor node to be unmarked before the actions.

We show the proofs for the `add` and `rmv` methods in Figures 55 and 56 respectively. Proof for the `ctn` method is the same as the lazy list. Below we mainly explain the verification of `add`. It is similar for `rmv`.

As usual, the linearization point for a successful `add` is at the time when the new node is linked to the list, *i.e.*, at line 22 for a successful `cas` in Figure 53. Thus, we insert **linself** at that line, as shown in Figure 55. But for an unsuccessful add, the LP is not static, whose location may depend on future behaviors. This is because when the `add` thread traverses the list, other threads could concurrently access the list. Thus even when `add` found the node was in the list in its traversal, its environment may have removed the node at the time when `add` returns **false**. But when `add` just read the node it is looking for in its traversal (such as at line 3 in Figure 53), the thread does not know whether the node would become marked or not when it reads its `mark` field (such as at line 5). Moreover, if the node is marked and also has been physically removed, the thread will restart the traversal, and the guessed LPs are all obsolete.

Thus in Figure 55, we insert **trylinself** at every first read of a node (*i.e.*, getting a pointer pointing to the node). We do not lose any chance when the current abstract set contains the node and from that point the concrete execution could return **false** in the future. We **commit** the original `ADD` operation when we are sure that we will restart or continue searching, and **commit** the (**end**, **false**) operation when we will return **false**. These auxiliary commands are highlighted in Figure 55. The proofs simply follow our inference rules. The main complexity is to distinguish whether a node currently visited is on list, or marked, or has been physically removed.

```
struct Node{ int val; Node *next; bool mark; };
```

```
locate(e) :
     local p, c, n, m, s;
  1  while (true) {
  2    p := Head;
  3    c := p.next;
  4    while (true) {
  5      (n, m) := c.(next, mark);
  6      while (m) {
  7        s := cas(&(p.(next, mark)), c, n, false, false);
  8        if (!s) continue line 1;
  9        c := n;
 10        (n, m) := c.(next, mark);
 11      }
 12      if (c.val >= e)
 13        return (p, c);
 14      p := c;
 15      c := n;
 16    }
 17  }

add(e) :
     local p, c, n;
 18  while (true) {
 19    (p, c) := locate(e);
 20    if (c.val != e) {
 21      n := cons(e, c, false);
 22      if (cas(&(p.(next, mark)), c, n, false, false))
 23        return true;
 24    } else {
 25      return false;
 26    }
 27  }
```

```
rmv(e) :
     local p, c, n, s;
 28  while (true) {
 29    (p, c) := locate(e);
 30    if (c.val = e) {
 31      n := c.next;
 32      s := cas(&(c.(next, mark)), n, n, false, true);
 33      if (!s)
 34        continue;
 35      cas(&(p.(next, mark)), c, n, false, false);
 36      return true;
 37    } else {
 38      return false;
 39    }
 40  }

ctn(e) :
     local c;
 41  c := Head;
 42  while (c.val < e) {
 43    c := c.next;
 44  }
 45  b := c.mark;
 46  if (!b && c.val = e)
 47    return true;
 48  else
 49    return false;
```

Implementation (Based on Herlihy and Shavit's Book and Errata)

Figure 53. Harris-Michael Lock-Free List

$$I \overset{\text{def}}{=} \exists A.\ \mathsf{ls}(\mathtt{Head}, A, \mathtt{null}) * \mathsf{s}(A) * \mathsf{Ds}(\mathtt{OutOps}) * \mathsf{garb}$$

$$\mathsf{N}(x,v,y,b) \overset{\text{def}}{=} x \mapsto (v,y,b) \qquad \mathsf{M}(x,v,y) \overset{\text{def}}{=} \mathsf{N}(x,v,y,\mathbf{true}) \qquad \mathsf{U}(x,v,y) \overset{\text{def}}{=} \mathsf{N}(x,v,y,\mathbf{false})$$

$$\mathsf{ls}(x,A,z) \overset{\text{def}}{=} (x = z \wedge A = \epsilon) \vee (x \neq z \wedge \exists v,y,b,A'.\ \mathsf{N}(x,v,y,b) * \mathsf{ls}(y,A',z) \wedge A = (v,b) :: A')$$

$$\mathsf{sorted}(A) \overset{\text{def}}{=} \begin{cases} \mathbf{true} & \text{if } A = \epsilon \vee A = (v,b) :: \epsilon \\ (v_1 < v_2) \wedge \mathsf{sorted}((v_2,b_2) :: A') & \text{if } A = (v_1,b_1) :: (v_2,b_2) :: A' \end{cases}$$

$$\mathsf{elems}(A) \overset{\text{def}}{=} \begin{cases} \emptyset & \text{if } A = \epsilon \\ \{v\} \cup \mathsf{elems}(A') & \text{if } A = (v,\mathbf{false}) :: A' \\ \mathsf{elems}(A') & \text{if } A = (v,\mathbf{true}) :: A' \end{cases}$$

$$\mathsf{s}(A) \overset{\text{def}}{=} \exists A'.\ (A = (\mathtt{MIN},\mathbf{false}) :: A' :: (\mathtt{MAX},\mathbf{false})) \wedge \mathsf{sorted}(A) \wedge (\mathtt{S} \Mapsto \mathsf{elems}(A')) \qquad \mathsf{garb} \overset{\text{def}}{=} \circledast_{x \in \mathtt{GN}}.\mathsf{M}(x,\_,\_)$$

$$\mathsf{d}(d,t,n,b) \overset{\text{def}}{=} d \mapsto (t,n,b) \wedge (b = \mathtt{UNDEF} \vee b = \mathbf{false})$$

$$\mathsf{notDone}(d,t,n) \overset{\text{def}}{=} \mathsf{d}(d,t,n,\mathtt{UNDEF}) * t \rightarrowtail (\mathtt{CTN}, n)$$

$$\mathsf{afterTrylin}(d,t,n) \overset{\text{def}}{=} \mathsf{d}(d,t,n,\mathbf{false}) * (t \rightarrowtail (\mathtt{CTN},n) \oplus t \rightarrowtail (\mathbf{end},\mathbf{false}))$$

$$\mathsf{D}(d,t,n) \overset{\text{def}}{=} \mathsf{notDone}(d,t,n) \vee \mathsf{afterTrylin}(d,t,n) \qquad \mathsf{Ds}(O) \overset{\text{def}}{=} \circledast_{d \in O}.\mathsf{D}(d,\_,\_)$$

$$R_{\mathsf{t}} \overset{\text{def}}{=} \bigvee_{\mathsf{t}' \neq \mathsf{t}} G_{\mathsf{t}'}$$

$$G_{\mathsf{t}} \overset{\text{def}}{=} [\mathsf{Add} \vee \mathsf{Mark} \vee \mathsf{Rmv} \vee \mathsf{AddOut}_{\mathsf{t}} \vee \mathsf{RmvOut}_{\mathsf{t}}]_I$$

$$\begin{aligned} \mathsf{Add} \overset{\text{def}}{=} {} & \exists x,y,z,p,u,v,w,b,S.\ (\mathsf{U}(x,u,y) * \mathsf{N}(y,w,z,b) * (\mathtt{S} \Mapsto S) \wedge (u < v < w)) \\ & \ltimes (\mathsf{U}(x,u,p) * \mathsf{U}(p,v,y) * \mathsf{N}(y,w,z,b) * (\mathtt{S} \Mapsto S \cup \{v\})) \end{aligned}$$

$$\mathsf{TrylinOut}(O,v) \overset{\text{def}}{=} \circledast_{d \in O}.(\exists t.\ \mathsf{D}(d,t,v) \ltimes \mathsf{afterTrylin}(d,t,v)) \qquad \text{provided } \exists O'.\ (\mathtt{OutOps} = O \uplus O') \wedge \forall d \in O'.\ \exists n.\ (n \neq v) \wedge \mathsf{D}(d,\_,n)$$

$$\mathsf{Mark} \overset{\text{def}}{=} \exists y,z,v,S,O.\ ((\mathsf{U}(y,v,z) * (\mathtt{S} \Mapsto S \cup \{v\}) \wedge (v < \mathtt{MAX})) \ltimes (\mathsf{M}(y,v,z) * (\mathtt{S} \Mapsto S))) * \mathsf{TrylinOut}(v,O)$$

$$\mathsf{Rmv} \overset{\text{def}}{=} \exists x,y,z,u,v,S_g.\ (\mathsf{U}(x,u,y) * \mathsf{M}(y,v,z) * (\mathtt{GN} = S_g) \wedge (v < \mathtt{MAX})) \ltimes (\mathsf{U}(x,u,z) * \mathsf{M}(y,v,z) * (\mathtt{GN} = S_g \cup \{y\}))$$

$$\mathsf{AddOut}_{\mathsf{t}} \overset{\text{def}}{=} \exists O,d.\ (\mathtt{OutOps} = O) \ltimes ((\mathtt{OutOps} = O \uplus \{d\}) * \mathsf{D}(d,\mathsf{t},\_))$$

$$\mathsf{RmvOut}_{\mathsf{t}} \overset{\text{def}}{=} \exists O,d.\ ((\mathtt{OutOps} = O \uplus \{d\}) * \mathsf{D}(d,\mathsf{t},\_)) \ltimes (\mathtt{OutOps} = O)$$

$$\mathsf{addMaynotin} \overset{\text{def}}{=} \mathsf{t} \rightarrowtail (\mathtt{ADD}, \mathsf{e})$$

$$\mathsf{addMayin} \overset{\text{def}}{=} \mathsf{t} \rightarrowtail (\mathtt{ADD}, \mathsf{e}) \oplus \mathsf{t} \rightarrowtail (\mathbf{end}, \mathbf{false})$$

$$\mathsf{rmvMayin} \overset{\text{def}}{=} \mathsf{t} \rightarrowtail (\mathtt{RMV}, \mathsf{e})$$

$$\mathsf{rmvMaynotin} \overset{\text{def}}{=} \mathsf{t} \rightarrowtail (\mathtt{RMV}, \mathsf{e}) \oplus \mathsf{t} \rightarrowtail (\mathbf{end}, \mathbf{false})$$

$$\mathsf{bound}(p,e) \overset{\text{def}}{=} \exists u.\ (I \wedge \mathsf{N}(p,u,\_,\_) * \mathsf{true}) * \mathsf{true} \wedge (u < e < \mathtt{MAX})$$

$$\mathsf{nodes2}(c,v,n,v') \overset{\text{def}}{=} (\mathsf{onlist2}(c,v,n,v') \vee (\mathsf{onlist}(c,v) \wedge \mathsf{notonlist}(n)) \vee (\mathsf{notonlist}(c) \wedge \mathsf{onlist}(n,v')) \vee (\mathsf{notonlist}(c) \wedge \mathsf{notonlist}(n))) \wedge v < v'$$

$$\mathsf{onlist}(c,v) \overset{\text{def}}{=} \exists x,b,A,B.\ \mathsf{ls}(\mathtt{Head},A,c) * \mathsf{N}(c,v,x,b) * \mathsf{ls}(x,B,\mathtt{null}) * \mathsf{s}(A :: (v,b) :: B) * \mathsf{Ds}(\mathtt{OutOps}) * \mathsf{garb}$$

$$\mathsf{onlistm}(c,v,n,v') \overset{\text{def}}{=} \exists x,b,A,B.\ \mathsf{ls}(\mathtt{Head},A,c) * \mathsf{M}(c,v,n) * \mathsf{N}(n,v',x,b) * \mathsf{ls}(x,B,\mathtt{null}) * \mathsf{s}(A :: (v,\mathbf{true}) :: (v',b) :: B) * \mathsf{Ds}(\mathtt{OutOps}) * \mathsf{garb}$$

$$\begin{aligned} & \mathsf{onlist2}(c,v,n,v') \overset{\text{def}}{=} \\ & \quad \exists x,y,A,B,C.\ \mathsf{ls}(\mathtt{Head},A,c) * \mathsf{N}(c,v,x,b) * \mathsf{ls}(x,B,n) * \mathsf{N}(n,v',y,b') * \mathsf{ls}(y,C,\mathtt{null}) * \mathsf{s}(A :: (v,b) :: B :: (v',b') :: C) * \mathsf{Ds}(\mathtt{OutOps}) * \mathsf{garb} \end{aligned}$$

$$\mathsf{onlistmax}(c,n) \overset{\text{def}}{=} \exists x,A.\ \mathsf{ls}(\mathtt{Head},A,c) * \mathsf{U}(c,\mathtt{MAX},\mathtt{null}) * (n = \mathtt{null}) * \mathsf{s}(A :: (\mathtt{MAX},\mathbf{false})) * \mathsf{Ds}(\mathtt{OutOps}) * \mathsf{garb}$$

$$\mathsf{notonlist}(c) \overset{\text{def}}{=} I \wedge (c \in \mathtt{GN})$$

Figure 54. Precise Invariant, Rely and Guarantee of Lock-Free List (for Thread t)

```
add(e):
  local p, c, n, m, s;
 $\{I * \mathsf{t} \rightarrowtail (\mathtt{ADD}, \mathsf{e}) \land (\mathtt{MIN} < \mathsf{e} < \mathtt{MAX})\}$
  while (true) {
    retry: p := Head;
    < c := p.next;
      if (e = c.val && !c.mark) trylinself; >
   $\{\exists v.\ (\mathsf{onlist}(\mathsf{c}, v) \lor \mathsf{notonlist}(\mathsf{c})) * ((\mathsf{e} \neq v \land \mathsf{addMaynotin}) \lor (\mathsf{e} = v \land \mathsf{addMayin})) \land \mathsf{bound}(\mathsf{p}, \mathsf{e})\}$
    while (true) {
      < (n, m) := c.(next, mark);
        if (!m && n != null && e = n.val && !n.mark) trylinself; >
     $\left\{\begin{array}{l} \mathsf{m} \land \exists v, v'.\ (\mathsf{onlistm}(\mathsf{c}, v, \mathsf{n}, v') \lor \mathsf{notonlist}(\mathsf{c}, v)) * ((\mathsf{e} \neq v \land \mathsf{addMaynotin}) \lor (\mathsf{e} = v \land \mathsf{addMayin})) \land \mathsf{bound}(\mathsf{p}, \mathsf{e}) \\ \lor \neg\mathsf{m} \land \mathsf{onlistmax}(\mathsf{c}, \mathsf{n}) * \mathsf{addMaynotin} \land \mathsf{bound}(\mathsf{p}, \mathsf{e}) \\ \lor \neg\mathsf{m} \land \exists v, v'.\ \mathsf{nodes2}(\mathsf{c}, v, \mathsf{n}, v') * (((\mathsf{e} \neq v \lor \mathsf{e} \neq v') \land \mathsf{addMaynotin}) \lor ((v = \mathsf{e} \lor \mathsf{e} = v') \land \mathsf{addMayin})) \land \mathsf{bound}(\mathsf{p}, \mathsf{e}) \end{array}\right\}$
      while (m) {
       $\{\exists v, v'.\ (\mathsf{onlistm}(\mathsf{c}, v, \mathsf{n}, v') \lor \mathsf{notonlist}(\mathsf{c}, v)) * ((\mathsf{e} \neq v \land \mathsf{addMaynotin}) \lor (\mathsf{e} = v \land \mathsf{addMayin})) \land \mathsf{bound}(\mathsf{p}, \mathsf{e})\}$
        < s := cas(&(p.(next, mark)), c, n, false, false);  if (s) GN := GN $\cup$ {c};
          if (s && e = c.val) commit(t $\rightarrowtail$ (ADD, e));  if (s && e = n.val) trylinself; >
       $\left\{\begin{array}{l} \mathsf{s} \land \exists v'.\ (\mathsf{onlist}(\mathsf{n}, v') \lor \mathsf{notonlist}(\mathsf{n})) * ((\mathsf{e} \neq v' \land \mathsf{addMaynotin}) \lor (\mathsf{e} = v' \land \mathsf{addMayin})) \land \mathsf{bound}(\mathsf{p}, \mathsf{e}) \\ \lor \neg\mathsf{s} \land I * (\mathsf{addMayin} \lor \mathsf{addMaynotin}) \land (\mathsf{e} < \mathtt{MAX}) \end{array}\right\}$
        if (!s) {
          commit(t $\rightarrowtail$ (ADD, e));
         $\{I * \mathsf{t} \rightarrowtail (\mathtt{ADD}, \mathsf{e}) \land (\mathsf{e} < \mathtt{MAX})\}$
          continue retry;
        }
        c := n;
        < (n, m) := c.(next, mark);
          if (!m && n != null && e = n.val && !n.mark) trylinself; >
      }
      if (c.val >= e)
        break;
     $\{\exists v, v'.\ \mathsf{nodes2}(\mathsf{c}, v, \mathsf{n}, v') * ((\mathsf{e} \neq v' \land \mathsf{addMaynotin}) \lor (\mathsf{e} = v' \land \mathsf{addMayin})) \land \mathsf{bound}(\mathsf{c}, \mathsf{e})\}$
      p := c;
      c := n;
    }
   $\left\{\begin{array}{l} \mathsf{onlistmax}(\mathsf{c}, \mathsf{n}) * \mathsf{addMaynotin} \land \mathsf{bound}(\mathsf{p}, \mathsf{e}) \\ \lor \exists v, v'.\ \mathsf{nodes2}(\mathsf{c}, v, \mathsf{n}, v') * ((\mathsf{e} < v \land \mathsf{addMaynotin}) \lor (\mathsf{e} = v \land \mathsf{addMayin})) \land \mathsf{bound}(\mathsf{p}, \mathsf{e}) \end{array}\right\}$
    if (c.val != e) {
     $\{\exists v.\ (\mathsf{onlistmax}(\mathsf{c}, \mathsf{n}) \lor \mathsf{onlist}(\mathsf{c}, v) \lor \mathsf{notonlist}(\mathsf{c})) * \mathsf{t} \rightarrowtail (\mathtt{ADD}, \mathsf{e}) \land \mathsf{bound}(\mathsf{p}, \mathsf{e}) \land \mathsf{e} < v\}$
      n := cons(e, c, false);
     $\{\exists v.\ (\mathsf{onlistmax}(\mathsf{c}, \mathsf{n}) \lor \mathsf{onlist}(\mathsf{c}, v) \lor \mathsf{notonlist}(\mathsf{c})) * \mathsf{U}(\mathsf{n}, \mathsf{e}, \mathsf{c}) * \mathsf{t} \rightarrowtail (\mathtt{ADD}, \mathsf{e}) \land \mathsf{bound}(\mathsf{p}, \mathsf{e}) \land \mathsf{e} < v\}$
      < s := cas(&(p.(next, mark)), c, n, false, false);
        if (s) linself; >
      if (!s)
       $\{I * \mathsf{t} \rightarrowtail (\mathtt{ADD}, \mathsf{e}) \land (\mathsf{e} < \mathtt{MAX})\}$
        continue;
     $\{I * \mathsf{t} \rightarrowtail (\mathbf{end}, \mathbf{true})\}$
      return true;
    } else {
      commit(t $\rightarrowtail$ (end, false));
     $\{I * \mathsf{t} \rightarrowtail (\mathbf{end}, \mathbf{false})\}$
      return false;
    }
  }
```

Figure 55. Proof Outline of Add of Lock-Free List for Thread t

```
IntSet GN;   //Auxiliary global variable for verification: removed nodes
IntSet OutOps;   //Auxiliary global variable for verification: thread descriptors for contains

rmv(e):
  local p, c, n, m, s;
 $\{I * \mathsf{t} \rightarrowtail (\mathtt{RMV}, \mathsf{e}) \wedge (\mathsf{e} < \mathtt{MAX})\}$
  while (true) {
    retry: p := Head;
    < c := p.next;
      if (e < c.val) trylinself; >
   $\{\exists v.\ (\mathsf{onlist}(\mathsf{c}, v) \vee \mathsf{notonlist}(\mathsf{c})) * ((\mathsf{e} < v \wedge \mathsf{rmvMaynotin}) \vee (v \le \mathsf{e} \wedge \mathsf{rmvMayin})) \wedge \mathsf{bound}(\mathsf{p}, \mathsf{e})\}$
    while (true) {
      < (n, m) := c.(next, mark);
        if (!m && n != null && c.val < e < n.val) trylinself; >
     $\left\{\begin{array}{l} \mathtt{m} \wedge \exists v, v'.\ (\mathsf{onlistm}(\mathsf{c}, v, \mathsf{n}, v') \vee \mathsf{notonlist}(\mathsf{c}, v)) * ((\mathsf{e} < v \wedge \mathsf{rmvMaynotin}) \vee (v \le \mathsf{e} \wedge \mathsf{rmvMayin})) \wedge \mathsf{bound}(\mathsf{p}, \mathsf{e}) \\ \vee \neg \mathtt{m} \wedge \mathsf{onlistmax}(\mathsf{c}, \mathsf{n}) * \mathsf{rmvMaynotin} \wedge \mathsf{bound}(\mathsf{p}, \mathsf{e}) \\ \vee \neg \mathtt{m} \wedge \exists v, v'.\ \mathsf{nodes2}(\mathsf{c}, v, \mathsf{n}, v') * (((\mathsf{e} < v \vee v < \mathsf{e} < v') \wedge \mathsf{rmvMaynotin}) \vee ((v = \mathsf{e} \vee v' \le \mathsf{e}) \wedge \mathsf{rmvMayin})) \wedge \mathsf{bound}(\mathsf{p}, \mathsf{e}) \end{array}\right\}$
      while (m) {
       $\{\exists v, v'.\ (\mathsf{onlistm}(\mathsf{c}, v, \mathsf{n}, v') \vee \mathsf{notonlist}(\mathsf{c}, v)) * ((\mathsf{e} < v \wedge \mathsf{rmvMaynotin}) \vee (v \le \mathsf{e} \wedge \mathsf{rmvMayin})) \wedge \mathsf{bound}(\mathsf{p}, \mathsf{e})\}$
        < s := cas(&(p.(next, mark)), c, n, false, false);  if (s) GN := GN $\cup$ {c};
          if (s && e < n.val) trylinself; >
       $\left\{\begin{array}{l} \mathtt{s} \wedge \exists v'.\ (\mathsf{onlist}(\mathsf{n}, v') \vee \mathsf{notonlist}(\mathsf{n})) * ((\mathsf{e} < v' \wedge \mathsf{rmvMaynotin}) \vee (v' \le \mathsf{e} \wedge \mathsf{rmvMayin})) \wedge \mathsf{bound}(\mathsf{p}, \mathsf{e}) \\ \vee \neg \mathtt{s} \wedge I * (\mathsf{rmvMayin} \vee \mathsf{rmvMaynotin}) \wedge (\mathsf{e} < \mathtt{MAX}) \end{array}\right\}$
        if (!s) {
          commit(t $\rightarrowtail$ (RMV, e));
         $\{I * \mathsf{t} \rightarrowtail (\mathtt{RMV}, \mathsf{e}) \wedge (\mathsf{e} < \mathtt{MAX})\}$
          continue retry;
        }
        c := n;
        < (n, m) := c.(next, mark);
          if (!m && n != null && c.val < e < n.val) trylinself; >
      }
      if (c.val >= e)
        break;
     $\{\exists v, v'.\ \mathsf{nodes2}(\mathsf{c}, v, \mathsf{n}, v') * ((v < \mathsf{e} < v' \wedge \mathsf{rmvMaynotin}) \vee (v' \le \mathsf{e} \wedge \mathsf{rmvMayin})) \wedge \mathsf{bound}(\mathsf{c}, \mathsf{e})\}$
      p := c;
      c := n;
    }
   $\left\{\begin{array}{l} \mathsf{onlistmax}(\mathsf{c}, \mathsf{n}) * \mathsf{rmvMaynotin} \wedge \mathsf{bound}(\mathsf{p}, \mathsf{e}) \\ \vee \exists v, v'.\ \mathsf{nodes2}(\mathsf{c}, v, \mathsf{n}, v') * ((\mathsf{e} < v \wedge \mathsf{rmvMaynotin}) \vee (v = \mathsf{e} \wedge \mathsf{rmvMayin})) \wedge \mathsf{bound}(\mathsf{p}, \mathsf{e}) \end{array}\right\}$
    if (c.val = e) {
     $\{(\mathsf{onlist}(\mathsf{c}, \mathsf{e}) \vee \mathsf{notonlist}(\mathsf{c})) * \mathsf{t} \rightarrowtail (\mathtt{RMV}, \mathsf{e}) \wedge \mathsf{bound}(\mathsf{p}, \mathsf{e})\}$
      n := c.next;
      < s := cas(&(c.(next, mark)), n, n, false, true);
        if (s) { linself;
           foreach d in OutOps
             if (d.arg = e) { d.res:= false; trylin(d.id); } } >
      if (!s)
       $\{I * \mathsf{t} \rightarrowtail (\mathtt{RMV}, \mathsf{e}) \wedge (\mathsf{e} < \mathtt{MAX})\}$
        continue;
     $\{(\mathsf{onlistm}(\mathsf{c}, \mathsf{e}, \mathsf{n}, \_) \vee \mathsf{notonlist}(\mathsf{c})) * \mathsf{t} \rightarrowtail (\textbf{end}, \textbf{true}) \wedge \mathsf{bound}(\mathsf{p}, \mathsf{e})\}$
      < s := cas(&(p.(next, mark)), c, n, false, false);  if (s) GN := GN $\cup$ {c}; >
     $\{I * \mathsf{t} \rightarrowtail (\textbf{end}, \textbf{true})\}$
      return true;
    } else {
      commit(t $\rightarrowtail$ (end, false));
     $\{I * \mathsf{t} \rightarrowtail (\textbf{end}, \textbf{false})\}$
      return false;
    }
  }
```

Figure 56. Proof Outline of Remove of Lock-Free List for Thread t

E.10 CCAS

As we mentioned in Section 6.3, CCAS (conditional compare-and-swap) [30] is a simplified version of the RDCSS algorithm which will show later. We have given the CCAS code in Figure 16. Here we formally define the invariant, the rely and the guarantee in Figure 57, and give the full proof in Figure 58 where we highlight the instrumented auxiliary commands.

To define the precise invariant, we introduce an auxiliary variable D and an auxiliary field status in each descriptor. Here D collects the garbage thread descriptors, which were once put into a but now is not in a anymore. The status field is like the auxiliary variable EPush in HSY stack (Appendix E.2). Roughly, it is used to make ownership transfer of abstract operations explicit. The field has three values: UNDEF, DONE and RET. When the descriptor is allocated (line 3 in Figure 16), its status is UNDEF. When the operation is finished (lines 15 and 17), the field is set to DONE. Note currently the finished abstract operation is still shared between threads. When the thread wants to return, it needs to set status to RET, and gets back the ownership of the abstract operation.

We formally define the precise invariant in Figure 57. It says, the shared state contains three parts. The first part invFlag is the flag bit at concrete and abstract levels. The second part is about a and the affiliated thread descriptor and abstract operation. It says either a is a value at both levels (aVal); or the concrete a contains a descriptor (aDesc), and the corresponding abstract operation should not have finished (notDone), or may have tried to be linearized and finished successfully (trySucc) or failed (tryFail), or may have tried to be linearized several times (by the thread itself or by the environment) and both success and failure are possible speculations (tryBoth). The last part garb contains the thread descriptors which were in a but are not anymore. Those descriptors must be marked as DONE or RET, and if the status field is DONE, the corresponding abstract operation is also available.

The guarantee defined in Figure 57 corresponds to the actions in the code. PlaceD corresponds to line 4 or 7 in Figure 16, which stores the thread descriptor in a and transfers both the descriptor and the corresponding abstract operation from the thread-local state to shared. TrylinSucc and TrylinFail correspond to line 13, which have been discussed in Section 6.3. RmvDSucc and RmvDFail correspond to line 15 and 17 respectively, which remove the descriptor from a and also commit the correct guesses. Finally, Done2Ret sets the status field of the descriptor from DONE to RET, and transfers the ownership of the abstract operation back to the thread.

We give the proof in Figure 58. The proof is almost straightforward, following the rely-guarantee-based inference rules. To simplify the proof (and help define the precise invariant), we add line 11 in Figure 58, which reclaims the descriptor when we are sure that it will not be used by any thread anymore. Actually, as indicated by our proof, the descriptor is locally owned by the thread at that time. Adding this line does not affect the correctness of the algorithm, since the algorithm assumes garbage collectors to reclaim those unreachable descriptors. We just make part of the garbage collection work explicit, when we have enough knowledge to reclaim the garbage by ourselves.

The interesting part is to reason about line 14 in Figure 58. Before that line, we have aDesc, which is roughly notDone $\vee$ trySucc $\vee$ tryFail $\vee$ tryBoth, as we mentioned. It says, the abstract operation has not been helped (notDone), or the environment may have helped try to linearize it (trySucc or tryFail or tryBoth). Note that the environment is uncertain, so all the four cases are possible (connected by $\vee$), in particular we may have the single speculation notDone. This ensures that we cannot cheat by imagining some non-existent environment threads to help linearize the operation. After line 14, we are sure that the guess endSucc must exist if we read a **true** flag, and the guess endFail must exist otherwise, since the current thread have tried to linearize the operation. Then we will never fail at the commit commands later.

$I \stackrel{\text{def}}{=} \mathsf{invFlag} * (\mathsf{aVal} \vee \mathsf{aDesc}(\_,\_,\_,\_)) * \mathsf{garb}$

$\mathsf{invFlag} \stackrel{\text{def}}{=} \exists b.\ (\texttt{flag} = b) * \texttt{flag} \Mapsto b$ $\qquad$ $\mathsf{garb} \stackrel{\text{def}}{=} \circledast_{d \in \texttt{D}}.(\exists t.\ \mathsf{end}(d,t,\_) \vee \mathsf{dRet}(d,t))$

$\mathsf{dDone}(d,t) \stackrel{\text{def}}{=} d \mapsto (t,\_,\_,\texttt{DONE})$ $\qquad$ $\mathsf{dRet}(d,t) \stackrel{\text{def}}{=} d \mapsto (t,\_,\_,\texttt{RET})$

$\mathsf{begin}(d,t,o,n) \stackrel{\text{def}}{=} d \mapsto (t,o,n,\texttt{UNDEF}) * t \rightarrowtail (\mathrm{CCAS},o,n)$ $\qquad$ $\mathsf{end}(d,t,r) \stackrel{\text{def}}{=} \mathsf{dDone}(d,t) * t \rightarrowtail (\mathbf{end},r)$

$\mathsf{aVal} \stackrel{\text{def}}{=} \exists v.\ (\mathtt{a} = v) * (\mathtt{a} \Mapsto v) \wedge \neg\mathsf{IsDesc}(v)$

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

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

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

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

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

$R_{\mathsf{t}} \stackrel{\text{def}}{=} \bigvee_{\mathsf{t}' \neq \mathsf{t}} G_{\mathsf{t}'}$

$G_{\mathsf{t}} \stackrel{\text{def}}{=} (\mathsf{SetFlag} \vee \mathsf{PlaceD}_{\mathsf{t}} \vee \mathsf{TrylinSucc} \vee \mathsf{TrylinFail} \vee \mathsf{RmvDSucc} \vee \mathsf{RmvDFail} \vee \mathsf{Done2Ret}_{\mathsf{t}} \vee \mathsf{Id}) * \mathsf{Id} \wedge (I \ltimes I)$

$\mathsf{SetFlag} \stackrel{\text{def}}{=} \mathsf{invFlag} \ltimes \mathsf{invFlag}$

$\mathsf{PlaceD}_{\mathsf{t}} \stackrel{\text{def}}{=} \exists v,d,o,n.\ ((\mathtt{a} = v) \wedge \neg\mathsf{IsDesc}(v)) \ltimes ((\mathtt{a} = d) * \mathsf{begin}(d,\mathsf{t},o,n))$

$\mathsf{TrylinSucc} \stackrel{\text{def}}{=} (\exists t,o,n.\ (\texttt{flag} * \mathsf{notDone}(t,o,n)) \ltimes (\texttt{flag} * \mathsf{endSucc}(t,o,n))) \oplus \mathsf{Id}$

$\mathsf{TrylinFail} \stackrel{\text{def}}{=} (\exists t,b,o,n.\ (\neg\texttt{flag} * \mathsf{notDone}(t,o,n)) \ltimes (\neg\texttt{flag} * \mathsf{endFail}(t,o))) \oplus \mathsf{Id}$

$\mathsf{RmvDSucc} \stackrel{\text{def}}{=} \exists d,t,o,n,S_d.\ ((\mathtt{a} = d) * d \mapsto (t,o,n,\texttt{UNDEF}) * (\texttt{D} = S_d) * (\mathsf{endSucc}(t,o,n) \oplus \mathsf{true}))$
$\qquad \ltimes ((\mathtt{a} = n) * d \mapsto (t,o,n,\texttt{DONE}) * (\texttt{D} = S_d \cup \{d\}) * \mathsf{endSucc}(t,o,n))$

$\mathsf{RmvDFail} \stackrel{\text{def}}{=} \exists d,t,o,n,S_d.\ ((\mathtt{a} = d) * d \mapsto (t,o,n,\texttt{UNDEF}) * (\texttt{D} = S_d) * (\mathsf{endFail}(t,o) \oplus \mathsf{true}))$
$\qquad \ltimes ((\mathtt{a} = o) * d \mapsto (t,o,n,\texttt{DONE}) * (\texttt{D} = S_d \cup \{d\}) * \mathsf{endFail}(t,o))$

$\mathsf{Done2Ret}_{\mathsf{t}} \stackrel{\text{def}}{=} \exists d.\ \mathsf{end}(d,\mathsf{t},\_) \ltimes \mathsf{dRet}(d,\mathsf{t})$

$\mathsf{aDescSucc}(b,d,t,o,n) \stackrel{\text{def}}{=} b \wedge (\mathtt{a} = d) * d \mapsto (t,o,n,\texttt{UNDEF}) * (\mathsf{trySucc}(t,o,n) \vee \mathsf{tryBoth}(t,o,n))$

$\mathsf{aDescFail}(b,d,t,o,n) \stackrel{\text{def}}{=} \neg b \wedge (\mathtt{a} = d) * d \mapsto (t,o,n,\texttt{UNDEF}) * (\mathsf{tryFail}(t,o,n) \vee \mathsf{tryBoth}(t,o,n))$

$\mathsf{aDescOther}(d) \stackrel{\text{def}}{=} \exists t,o.\ (\mathsf{aDesc}(d,t,o,\_) \vee \mathsf{end}(d,t,o) \vee \mathsf{dRet}(d,t)) \wedge t \neq \texttt{cid}$

Figure 57. Precise Invariant, Rely and Guarantee of CCAS (for thread t)

$$\mathsf{casaSucc} \overset{\text{def}}{=} (\mathtt{r} = \mathtt{o}) * (I \wedge ((\mathsf{aDesc}(\mathtt{d}, \mathtt{cid}, \mathtt{o}, \mathtt{n}) \vee \mathsf{end}(\mathtt{d}, \mathtt{cid}, \mathtt{o})) * \mathsf{true}))$$
$$\mathsf{casaFailVal} \overset{\text{def}}{=} (\mathtt{r} \neq \mathtt{o} \wedge \neg \mathsf{IsDesc}(\mathtt{r})) * I * \mathsf{end}(\mathtt{d}, \mathtt{cid}, \mathtt{r})$$
$$\mathsf{casaFailDesc} \overset{\text{def}}{=} (I \wedge \mathsf{aDescOther}(\mathtt{r}) * \mathsf{true}) * \mathsf{begin}(\mathtt{d}, \mathtt{cid}, \mathtt{o}, \mathtt{n})$$

```
CCAS(o, n):
    local r, d;
   $\{ I * \mathtt{cid} \rightarrowtail (\mathsf{CCAS}, \mathtt{o}, \mathtt{n}) \}$
    d := cons(cid, o, n, UNDEF);
   $\{ I * \mathsf{begin}(\mathtt{d}, \mathtt{cid}, \mathtt{o}, \mathtt{n}) \}$
    < r := cas(&a, o, d);  if (r != o && !IsDesc(r)) linself; >
   $\{ \mathsf{casaSucc} \vee \mathsf{casaFailVal} \vee \mathsf{casaFailDesc} \}$
    while (IsDesc(r)) {
     $\{ \mathsf{casaFailDesc} \}$
      Complete(r);
     $\{ I * \mathsf{begin}(\mathtt{d}, \mathtt{cid}, \mathtt{o}, \mathtt{n}) \}$
      < r := cas(&a, o, d);  if (r != o && !IsDesc(r)) linself; >
    }
   $\{ \mathsf{casaSucc} \vee \mathsf{casaFailVal} \}$
    if (r = o) {
     $\{ \mathsf{casaSucc} \}$
      Complete(d);
     $\{ I \wedge \mathsf{end}(\mathtt{d}, \mathtt{cid}, \mathtt{r}) * \mathsf{true} \}$
      d.status := RET;
     $\{ (I \wedge \mathsf{dRet}(\mathtt{d}, \mathtt{cid}) * \mathsf{true}) * \mathtt{cid} \rightarrowtail (\mathbf{end}, \mathtt{r}) \}$
    } else {
     $\{ I * \mathsf{end}(\mathtt{d}, \mathtt{cid}, \mathtt{r}) \}$
      dispose(d);
     $\{ I * \mathtt{cid} \rightarrowtail (\mathbf{end}, \mathtt{r}) \}$
    }
   $\{ I * \mathtt{cid} \rightarrowtail (\mathbf{end}, \mathtt{r}) \}$
    return r;


Complete(d):
    local v, s;
   $\{ I \wedge (\mathsf{aDesc}(\mathtt{d}, t, o, n) \vee \mathsf{end}(\mathtt{d}, t, o) \vee (\mathsf{dRet}(\mathtt{d}, t) \wedge t \neq \mathtt{cid})) * \mathsf{true} \}$
    < v := flag;  if (a = d) trylin(d.id); >
   $\{ I \wedge (\mathsf{aDescSucc}(\mathtt{v}, \mathtt{d}, t, o, n) \vee \mathsf{aDescFail}(\mathtt{v}, \mathtt{d}, t, o, n) \vee \mathsf{end}(\mathtt{d}, t, o) \vee (\mathsf{dRet}(\mathtt{d}, t) \wedge t \neq \mathtt{cid})) * \mathsf{true} \}$
    if (v)
     $\{ I \wedge (\mathsf{aDescSucc}(\mathtt{v}, \mathtt{d}, t, o, n) \vee \mathsf{end}(\mathtt{d}, t, o) \vee (\mathsf{dRet}(\mathtt{d}, t) \wedge t \neq \mathtt{cid})) * \mathsf{true} \}$
      < s := cas(&a, d, d.n);  if (s = d) { d.status := DONE; D := $\mathtt{D} \cup \{\mathtt{d}\}$;  commit($\mathtt{d.id} \rightarrowtail (\mathsf{end}, \mathtt{d.o}) * \mathtt{a} \Rrightarrow \mathtt{d.n}$); } >
    else
     $\{ I \wedge (\mathsf{aDescFail}(\mathtt{v}, \mathtt{d}, t, o, n) \vee \mathsf{end}(\mathtt{d}, t, o) \vee (\mathsf{dRet}(\mathtt{d}, t) \wedge t \neq \mathtt{cid})) * \mathsf{true} \}$
      < s := cas(&a, d, d.o);  if (s = d) { d.status := DONE; D := $\mathtt{D} \cup \{\mathtt{d}\}$;  commit($\mathtt{d.id} \rightarrowtail (\mathsf{end}, \mathtt{d.o}) * \mathtt{a} \Rrightarrow \mathtt{d.o}$); } >
   $\{ I \wedge (\mathsf{end}(\mathtt{d}, t, o) \vee (\mathsf{dRet}(\mathtt{d}, t) \wedge t \neq \mathtt{cid})) * \mathsf{true} \}$
```

Figure 58. Proof Outline of CCAS for Thread `cid`

E.11 RDCSS

RDCSS (restricted double-compare single-swap) is part of Harris *et al.*'s MCAS algorithm [12]. We show its code in Figure 59. It manipulates two sets of memory cells: the set C is the control section, and the set A is the data section. RDCSS takes five parameters: a control location c in C, a data location a in A, an expected value e for c, an expected old value o for a, and a new value n for a. It wants to atomically update a's value to n, if both c and a contain the expected values e and o respectively; otherwise it does nothing. It returns a's old value. The code is similar to CCAS. The object also provides a method RDCSSRead, which reads from a in A. Note RDCSSRead will first complete the pending RDCSS operation if a descriptor for that operation is in a.

As in CCAS, we add an auxiliary variable D for collecting the garbage descriptors and an auxiliary field status in a descriptor for indicating the ownership of the descriptor (whether it is shared or thread-local). Both are write-only and will not affect the executions of the implementation.

The proof is similar to CCAS's proof. We define the precise invariant, the rely and the guarantee conditions in Figure 60, and outline the proof (with the instrumented auxiliary commands being highlighted) in Figure 61.

The precise invariant I defined in Figure 60 says, a shared state consists of three parts, the first part invC is the control section at both the concrete and abstract levels, the last part garb is for the garbage descriptors and abstract operations which have been finished but have not returned, and the remaining part is for the data section. Similar to the precise invariant for CCAS in Figure 57, each data a is a value at both levels (aVal); or the concrete a contains a descriptor (aDesc), and the corresponding abstract operation should not have finished (notDone), or may have tried to be linearized and finished successfully (trySucc) or failed (tryFail), or may have tried to be linearized several times (by the thread itself or by the environment) and both success and failure are possible speculations (tryBoth). The rely and the guarantee are similar to those for CCAS. Note now we are considering sets of locations rather than a single fixed flag and a.

The proof in Figure 61 is quite similar to CCAS, except that now we also take c and a as arguments.

```
struct ThrdDesc {
  int id;
  int c, a, e, o, n;
}

RDCSS(c, a, e, o, n) :
    local r, d;
    d := cons(cid, c, a, e, o, n);
    r := cas(a, o, d);
    while (IsDesc(r)) {
      Complete(r);
      r := cas(a, o, d);
    }
    if (r = o) {
      Complete(d);
    }
    return r;

Complete(d) :
    local v;
    v := [d.c];
    if (v = d.e)
      cas(d.a, d, d.n);
    else
      cas(d.a, d, d.o);

RDCSSRead(a) :
    local r;
    r := [a];
    while (IsDesc(r)) {
      Complete(r);
      r := [a];
    }
    return r;

WriteC(c, n) :
    [c] := n;
```

Figure 59. RDCSS Code

$I \stackrel{\text{def}}{=} \mathsf{invC} * (\circledast_{a \in A}.(\mathsf{aVal}(a) \vee \mathsf{aDesc}(\_,\_,a,\_))) * \mathsf{garb}$

$\mathsf{invC} \stackrel{\text{def}}{=} \circledast_{c \in C}.\ \mathsf{invc}(c) \qquad \mathsf{invc}(c) \stackrel{\text{def}}{=} \exists v.\ (c \mapsto v) * ([c] \Mapsto v)$

$\mathsf{garb} \stackrel{\text{def}}{=} \circledast_{d \in \mathtt{D}}.(\exists t.\ \mathsf{end}(d,t,\_) \vee \mathsf{dRet}(d,t)) \qquad \mathsf{dDone}(d,t) \stackrel{\text{def}}{=} d \mapsto (t,\_,\_,\_,\_,\_,\mathtt{DONE}) \qquad \mathsf{dRet}(d,t) \stackrel{\text{def}}{=} d \mapsto (t,\_,\_,\_,\_,\_,\mathtt{RET})$

$\mathsf{begin}(d,t,c,a,e,o,n) \stackrel{\text{def}}{=} d \mapsto (t,c,a,e,o,n,\mathtt{UNDEF}) * t \rightarrowtail (\mathrm{RDCSS},c,a,e,o,n) \qquad \mathsf{end}(d,t,r) \stackrel{\text{def}}{=} \mathsf{dDone}(d,t) * t \rightarrowtail (\mathbf{end},r)$

$\mathsf{aVal}(a) \stackrel{\text{def}}{=} \exists v.\ (a \mapsto v) * ([a] \Mapsto v) \wedge \neg \mathsf{IsDesc}(v)$

$\mathsf{aDesc}(d,t,c,a,e,o,n) \stackrel{\text{def}}{=}$
$\quad (a \mapsto d) * d \mapsto (t,c,a,e,o,n,\mathtt{UNDEF}) * (\mathsf{notDone}(t,c,a,e,o,n) \vee \mathsf{trySucc}(t,c,a,e,o,n) \vee \mathsf{tryFail}(t,c,a,e,o,n) \vee \mathsf{tryBoth}(t,c,a,e,o,n))$

$\mathsf{aDesc}(d,t,a,o) \stackrel{\text{def}}{=} \mathsf{aDesc}(d,t,\_,a,\_,o,\_)$

$\mathsf{notDone}(t,c,a,e,o,n) \stackrel{\text{def}}{=} t \rightarrowtail (\mathrm{RDCSS},c,a,e,o,n) * ([a] \Mapsto o)$

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

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

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

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

$R_{\mathsf{t}} \stackrel{\text{def}}{=} \bigvee_{\mathsf{t}' \neq \mathsf{t}} G_{\mathsf{t}'}$

$G_{\mathsf{t}} \stackrel{\text{def}}{=} [\mathsf{WriteC} \vee \mathsf{PlaceD}_{\mathsf{t}} \vee \mathsf{TrylinSucc} \vee \mathsf{TrylinFail} \vee \mathsf{RmvDSucc} \vee \mathsf{RmvDFail} \vee \mathsf{Done2Ret}_{\mathsf{t}}]_I$

$\mathsf{WriteC} \stackrel{\text{def}}{=} \exists c \in C.\ \mathsf{invc}(c) \ltimes \mathsf{invc}(c)$

$\mathsf{PlaceD}_{\mathsf{t}} \stackrel{\text{def}}{=} \exists v,d,a \in A.\ ((a \mapsto v) \wedge \neg \mathsf{IsDesc}(v)) \ltimes ((a \mapsto d) * \mathsf{begin}(d,\mathsf{t},\_,a,\_,\_,\_))$

$\mathsf{TrylinSucc} \stackrel{\text{def}}{=} (\exists t, c \in C, a \in A, e, o, n.\ ((c \mapsto e) * \mathsf{notDone}(t,c,a,e,o,n)) \ltimes ((c \mapsto e) * \mathsf{endSucc}(t,a,o,n))) \oplus \mathsf{Id}$

$\mathsf{TrylinFail} \stackrel{\text{def}}{=} (\exists t, c \in C, a \in A, e, o, n, v.\ ((c \mapsto v) * \mathsf{notDone}(t,c,a,e,o,n) \wedge v \neq e) \ltimes ((c \mapsto v) * \mathsf{endFail}(t,a,o))) \oplus \mathsf{Id}$

$\mathsf{RmvDSucc} \stackrel{\text{def}}{=} \exists d,t,a \in A, o, n, S_d.\ ((a \mapsto d) * d \mapsto (t,\_,a,\_,o,n,\mathtt{UNDEF}) * (\mathtt{D} = S_d) * (\mathsf{endSucc}(t,a,o,n) \oplus \mathsf{true}))$
$\qquad \ltimes ((a \mapsto n) * \mathsf{dDone}(d,t) * (\mathtt{D} = S_d \cup \{d\}) * \mathsf{endSucc}(t,a,o,n))$

$\mathsf{RmvDFail} \stackrel{\text{def}}{=} \exists d,t,a \in A, o, S_d.\ ((a \mapsto d) * d \mapsto (t,\_,a,\_,o,\_,\mathtt{UNDEF}) * (\mathtt{D} = S_d) * (\mathsf{endFail}(t,a,o) \oplus \mathsf{true}))$
$\qquad \ltimes ((a \mapsto o) * \mathsf{dDone}(d,t) * (\mathtt{D} = S_d \cup \{d\}) * \mathsf{endFail}(t,a,o))$

$\mathsf{Done2Ret}_{\mathsf{t}} \stackrel{\text{def}}{=} \exists d.\ \mathsf{end}(d,\mathsf{t},\_) \ltimes \mathsf{dRet}(d,\mathsf{t})$

$\mathsf{aDescSucc}(v,d,t,a,o) \stackrel{\text{def}}{=}$
$\quad \exists c,e,n.\ (v = e) * (a \mapsto d) * d \mapsto (t,c,a,e,o,n,\mathtt{UNDEF}) * (\mathsf{trySucc}(t,c,a,e,o,n) \vee \mathsf{tryBoth}(t,c,a,e,o,n))$

$\mathsf{aDescFail}(v,d,t,a,o) \stackrel{\text{def}}{=}$
$\quad \exists c,e,n.\ (v \neq e) * (a \mapsto d) * d \mapsto (t,c,a,e,o,n,\mathtt{UNDEF}) * (\mathsf{tryFail}(t,c,a,e,o,n) \vee \mathsf{tryBoth}(t,c,a,e,o,n))$

$\mathsf{aDescOther}(d,a) \stackrel{\text{def}}{=} \exists t,o.\ (\mathsf{aDesc}(d,t,a,o) \vee \mathsf{end}(d,t,o) \vee \mathsf{dRet}(d,t)) \wedge t \neq \mathtt{cid}$

Figure 60. Precise Invariant, Rely and Guarantee of RDCSS (for thread t)

$$\mathsf{casaSucc} \stackrel{\text{def}}{=} (\mathsf{r} = \mathsf{o}) * (I \wedge (\mathsf{aDesc}(\mathsf{d}, \mathsf{cid}, \mathsf{a}, \mathsf{o}) \vee \mathsf{end}(\mathsf{d}, \mathsf{cid}, \mathsf{o})) * \mathsf{true})$$
$$\mathsf{casaFailVal} \stackrel{\text{def}}{=} (\mathsf{r} \neq \mathsf{o} \wedge \neg \mathsf{IsDesc}(\mathsf{r})) * I * \mathsf{end}(\mathsf{d}, \mathsf{cid}, \mathsf{r})$$
$$\mathsf{casaFailDesc} \stackrel{\text{def}}{=} (I \wedge \mathsf{aDescOther}(\mathsf{r}, \mathsf{a}) * \mathsf{true}) * \mathsf{begin}(\mathsf{d}, \mathsf{cid}, \mathsf{c}, \mathsf{a}, \mathsf{e}, \mathsf{o}, \mathsf{n})$$

```
RDCSS(c, a, e, o, n):
    local r, d;
   $\{ I * \mathsf{cid} \rightarrowtail (\mathrm{RDCSS}, \mathsf{c}, \mathsf{a}, \mathsf{e}, \mathsf{o}, \mathsf{n}) \}$
    d := cons(cid, c, a, e, o, n, UNDEF);
   $\{ I * \mathsf{begin}(\mathsf{d}, \mathsf{cid}, \mathsf{c}, \mathsf{a}, \mathsf{e}, \mathsf{o}, \mathsf{n}) \}$
    < r := cas(a, o, d); if (r != o && !IsDesc(r)) linself; >
   $\{ \mathsf{casaSucc} \vee \mathsf{casaFailVal} \vee \mathsf{casaFailDesc} \}$
    while (IsDesc(r)) {
      $\{ \mathsf{casaFailDesc} \}$
      Complete(r);
      $\{ I * \mathsf{begin}(\mathsf{d}, \mathsf{cid}, \mathsf{c}, \mathsf{a}, \mathsf{e}, \mathsf{o}, \mathsf{n}) \}$
      < r := cas(a, o, d); if (r != o && !IsDesc(r)) linself; >
    }
   $\{ \mathsf{casaSucc} \vee \mathsf{casaFailVal} \}$
    if (r = o) {
      $\{ \mathsf{casaSucc} \}$
      Complete(d);
      $\{ I \wedge \mathsf{end}(\mathsf{d}, \mathsf{cid}, \mathsf{r}) * \mathsf{true} \}$
      d.status := RET;
      $\{ (I \wedge \mathsf{dRet}(\mathsf{d}, \mathsf{cid}) * \mathsf{true}) * \mathsf{cid} \rightarrowtail (\mathbf{end}, \mathsf{r}) \}$
    } else {
      $\{ I * \mathsf{end}(\mathsf{d}, \mathsf{cid}, \mathsf{r}) \}$
      dispose(d);
      $\{ I * \mathsf{cid} \rightarrowtail (\mathbf{end}, \mathsf{r}) \}$
    }
   $\{ I * \mathsf{cid} \rightarrowtail (\mathbf{end}, \mathsf{r}) \}$
    return r;


Complete(d):
    local v, s;
   $\{ I \wedge (\mathsf{aDesc}(\mathsf{d}, t, a, o) \vee \mathsf{end}(\mathsf{d}, t, o) \vee (\mathsf{dRet}(\mathsf{d}, t) \wedge t \neq \mathsf{cid})) * \mathsf{true} \}$
    < v := [d.c]; if ([d.a] = d) trylin(d.id); >
   $\{ I \wedge (\mathsf{aDescSucc}(\mathsf{v}, \mathsf{d}, t, a, o) \vee \mathsf{aDescFail}(\mathsf{v}, \mathsf{d}, t, a, o) \vee \mathsf{end}(\mathsf{d}, t, o) \vee (\mathsf{dRet}(\mathsf{d}, t) \wedge t \neq \mathsf{cid})) * \mathsf{true} \}$
    if (v = d.e)
      $\{ I \wedge (\mathsf{aDescSucc}(\mathsf{v}, \mathsf{d}, t, a, o) \vee \mathsf{end}(\mathsf{d}, t, o) \vee (\mathsf{dRet}(\mathsf{d}, t) \wedge t \neq \mathsf{cid})) * \mathsf{true} \}$
      < s := cas(d.a, d, d.n); if (s = d) { d.status := DONE; D := $\mathsf{D} \cup \{\mathsf{d}\}$; commit($\mathsf{d.id} \rightarrowtail (\mathsf{end}, \mathsf{d.o}) * \mathsf{d.a} \Mapsto \mathsf{d.n}$); } >
    else
      $\{ I \wedge (\mathsf{aDescFail}(\mathsf{v}, \mathsf{d}, t, a, o) \vee \mathsf{end}(\mathsf{d}, t, o) \vee (\mathsf{dRet}(\mathsf{d}, t) \wedge t \neq \mathsf{cid})) * \mathsf{true} \}$
      < s := cas(d.a, d, d.o); if (s = d) { d.status := DONE; D := $\mathsf{D} \cup \{\mathsf{d}\}$; commit($\mathsf{d.id} \rightarrowtail (\mathsf{end}, \mathsf{d.o}) * \mathsf{d.a} \Mapsto \mathsf{d.o}$); } >
   $\{ I \wedge (\mathsf{end}(\mathsf{d}, t, o) \vee (\mathsf{dRet}(\mathsf{d}, t) \wedge t \neq \mathsf{cid})) * \mathsf{true} \}$


RDCSSRead(a):
    local r;
   $\{ I * \mathsf{cid} \rightarrowtail (\mathrm{RDCSSRead}, \mathsf{a}) \}$
    < r := [a]; if (!IsDesc(r)) linself; >
   $\{ (I \wedge \mathsf{aDescOther}(\mathsf{r}, \mathsf{a}) * \mathsf{true}) * \mathsf{cid} \rightarrowtail (\mathrm{RDCSSRead}, \mathsf{a}) \vee (\neg \mathsf{IsDesc}(\mathsf{r}) \wedge I * \mathsf{cid} \rightarrowtail (\mathbf{end}, \mathsf{r})) \}$
    while (IsDesc(r)) {
      $\{ (I \wedge \mathsf{aDescOther}(\mathsf{r}, \mathsf{a}) * \mathsf{true}) * \mathsf{cid} \rightarrowtail (\mathrm{RDCSSRead}, \mathsf{a}) \}$
      Complete(r);
      $\{ I * \mathsf{cid} \rightarrowtail (\mathrm{RDCSSRead}, \mathsf{a}) \}$
      < r := [a]; if (!IsDesc(r)) linself; >
    }
   $\{ I * \mathsf{cid} \rightarrowtail (\mathbf{end}, \mathsf{r}) \}$
    return r;
```

Figure 61. Proof Outline of RDCSS for Thread `cid`