Simplifying Linearizability Proofs with Reduction and Abstraction

Tayfun Elmas
Koç University, İstanbul, Turkey
telmas@ku.edu.tr

Shaz Qadeer
Microsoft Research, Redmond, WA
qadeer@microsoft.com

Ali Sezgin
Koç University, İstanbul, Turkey
asezgin@ku.edu.tr

Omer Subasi
Koç University, İstanbul, Turkey
osubasi@ku.edu.tr

Serdar Tasiran
Koç University, İstanbul, Turkey
stasiran@ku.edu.tr

Abstract

The typical proof of linearizability establishes an abstraction map from the concurrent program to a sequential specification, and identifies the commit points of operations. If the concurrent program uses fine-grained concurrency and complex synchronization, constructing such a proof is difficult. We propose a sound proof system that significantly simplifies the reasoning about linearizability. Linearizability is proved by transforming an implementation into its specification within this proof system. The proof system combines reduction and abstraction, which increase the granularity of atomic actions, with variable introduction and hiding, which relate the synchronization mechanism of the implementation to that of the specification. We construct the abstraction map incrementally, and eliminate the need to reason about the location of commit points in the implementation. We have implemented our method in the QED verifier and demonstrate its effecacy and practicality on several highly-concurrent examples from the literature.

1. Introduction

Linearizability is a well-known correctness criterion for concurrent data-structure implementations [10]. It relates a concurrent implementation, denoted *Impl*, to a sequential specification, denoted *Spec*, by the condition that every concurrent operation *op* of *Impl* takes effect atomically between its call and return points, where the correct effect is described by a sequential operation op' in *Spec*.

The typical proof of linearizability establishes an *abstraction map*, from *Impl*-states to *Spec*-states [1], and shows that only one action of *op*, called the *commit action*, is mapped to op', and other actions are mapped to stuttering (identity) transitions in *Spec*. Identifying the commit action becomes nontrivial when *op* is written in terms of many small actions that make visible changes to the state. While the abstraction map relates *Impl*-states to *Spec*-states, it must also filter out the effects of the partially completed operations on the state except for the commit action. This possibly involves completing partial operations or rolling back the effects of these actions back to a clean state [19]. Under fine-grained concurrency and complicated concurrency control, defining this abstraction map requires considerable expertise.

In this paper, we propose a sound proof system that simplifies reasoning about linearizability. Our key idea is rewriting the program with larger atomic actions. In [5], we showed that by interleaving reduction with abstraction, we can increase atomicity to the point that assertions in a concurrent program can be verified by sequential reasoning. In this work, we extend this approach with new proof rules, and argue that program rewriting guided by atomicity is an effective method to cope with linearizability proofs as well.

We prove that *Impl* is linearizable with respect to *Spec*, by transforming *Impl* up to *Spec* in a sequence of phases. In the *reduction* phase, we alternate reduction and abstraction to mark a set of sequentially composed actions as atomic. These actions are collected together, and the effects of the interleaving threads are eliminated. In the *refinement* phase, we couple variable introduction and a new rule, variable hiding, in order to make the code closer to the specification. These techniques provide us with the ability to *syntactically* relate implementation of a data structure to a specification with a different representation.

Our method has several key advantages. First, we support the incremental construction of the abstraction map with techniques that increase atomicity. Each refinement phase towards the specification is performed only after reaching a proper atomicity level. Second, we do not require the identification of the commit points. This is especially helpful when the commit point is determined at runtime depending on thread interleavings. Third, we provide the soundness guarantee that, the proof transformations, while they grow atomic blocks, preserve the behaviors of the original program that are relevant to linearizability. Thus, our method is a useful and sound complement to other proof techniques.

The refinement phase also helps to improve reduction, by eliminating superficial conflicts. Two equivalent operations might conflict on low-level (implementation) variables but this does not necessarily correspond to real conflicts in terms of the final specification. Our solution to this issue indirectly introduces a semantic hierarchy into mover checks in reduction, which is not particular to linearizability and is likely to be useful in any kind of reduction proof.

We have implemented our method in the QED verifier. We demonstrate the effecacy and practicality of our method by proving

[Copyright notice will appear here once 'preprint' option is removed.]

linearizability of several tricky examples from the literature. All proofs are available online and reproducible using QED.

This paper makes the following contributions:

– Improving the proof system in [5] with variable introduction and hiding

– Proving that complementing other methods with our transformations is sound

– Proving that transforming *Impl* to *Spec* guarantees linearizability of *Impl*

– Implementing the proposed method in QED

– Proving linearizability of highly-concurrent data structures in QED

1.1 Related work

Refinement between a concurrent program and its sequential specification is well-studied [1, 11, 12, 13]. Previous work showed that, under certain conditions, auxiliary variables enable construction of an abstraction map for every refinement relation [1, 11]. However, applying these techniques in practice remains a challenge. [19] used a complex abstraction map, called *aggregation function*, that completes the atomic transactions that are committed but not yet finished. The refinement proofs in [10, 9, 7, 3], despite being supported by automated proof checkers, all require manual guidance for the derivation of the proof. Recently, [21] provided a tool that automates the derivation of the proof using shape abstraction. To our knowledge, its automating ability is limited to linked-list based data structures and it still requires identification of the possible commit points.

Owicki-Gries [18, 10] and rely-guarantee [22] methods have been used in refinement proofs. However, in the case of fine-grained concurrency, deriving the proof obligations in both approaches requires expertise. The idea of local reasoning is exploited by separation logic [2] which is not particularly useful for shared objects with high level of interference. In these cases, we show that abstraction is an important tool to reduce the effects of interference.

Wang and Stoller [23] statically prove linearizability of the program using its sequentially executed version as the specification. Their notion of atomicity is defined over a fixed set of primitives, which is limited in the case of superficial conflicts. On the other hand, our notion of atomicity is more general and supported by abstraction to prove atomicity even under high level of interference. They provided hand-crafted proofs for several non-blocking algorithms, and our proofs are mechanically checked. In [8], Groves gives a hand-proof of the linearizability of the nonblocking queue, by reducing executions the fine-grained program to its sequential version. His use of reduction is non-incremental, and must consider the commutativity of each action by doing a global reasoning, while our reasoning is local.

1.2 Outline of the paper

Section 2 complements this section by pointing out strengths of our approach over classical proofs of linearizability using an example. Section 3 presents the defintions underlying our soundness theorem. The proof transformations are explained in Section 4 and the soundness theorems are given in Section 5. Section 6 concludes by describing our implementation and illustrates key points of a proof using an example.

2. Motivation and overview

Our running example is a multiset of integers. Figure 1 shows the concurrent implementation (*Impl*), and the sequential specification (*Spec*), of `InsertPair` and `LookUp` operations[1]. The instruction `assume` ϕ blocks until ϕ becomes `true`, and `havoc x` assigns a nondeterminstic value to `x`. Our goal is to verify linearizability of *Impl* with respect to a specification given using the variable `S`. `S` maps each integer to its cardinality in the multiset. Initially, `S` is empty, so `S[x]==0` for every integer `x`.

Impl contains an array `M` of slots each storing one element of the multiset. The `elt` field stores the element, and the `stt` field indicates the status of the slot. `M[i].elt` is considered to be in the multiset only when `M[i].stt==full`. The atomic `FindSlot` operation[2] allocates an empty slot by setting its `stt` field to `reserved`, and returns its index. `FindSlot` fails and returns `-1` if it cannot find any empty slot. The lock of each slot is acquired and released separately by `lock` and `unlock` operations, respectively.

2.1 Challenges in a typical refinement proof for multiset

Abstraction maps and commit points. In the following, we envision a refinement proof for multiset using techniques in the literature, and highlight common difficulties. We then illustrate how our proposed approach alleviates them.

Many techniques work by first selecting a commit point in every operation. The most likely choice for the commit point for `InsertPair` is line 16, since releasing the first lock makes the inserted element visible to other threads. Consider an abstraction map from *Impl* to *Spec* and suppose that line 16 or `InsertPair` is executed by *Impl*. This transition must be mapped to a single transition that increments `S[x]` and `S[y]` atomically. As a first try, let us consider the simple abstraction map below. (`|A|` denotes the cardinality of the set `A`).

```
S[x]==|{ i | 0<=i<N && M[i].elt==x && M[i].stt==full }|
```

This map does not work with this choice of commit point, because when lines 14 and 15 of `InsertPair` are executed, `S[x]` and `S[y]` are incremented, but the execution has not reached the commit point yet. In addition, the updates that are propagated to `S` are not atomic. Our next, slightly more sophisticated map below does not update `S[x]` and `S[x]` while the locks for these cells are held. (`HeldBy(M[i],t)` is `true` when thread `t` is holding the lock of `M[i]`):

```
S[x]==|{ i | 0<=i<N && M[i].elt==x && M[i].stt==full
             && !HeldBy(M[i],t) }|
```

The problem with this map is that every slot locked by a thread would be excluded from `S`. As a result, at line 16 (the commit point) the map would increment `S[x]` but not `S[y]` since `M[j]` is still locked. Thus, this map still does not accomplish the atomic specification state update we are after. The right map has to complete this partial update at the commit point by incrementing `S[y]` as well although the lock of `M[j]` is still held.

We next try different selections of commit points: lines 14, 15 or 17. For each of these choices, in order to produce the intended specification state and avoid non-atomic updates to it, an abstraction map must "roll back" effects of executions of `InsertPair` that have not reached their commit point, and must "complete" the effects of others that are past their commit point but have not yet finished. To accomplish this, the map must refer to not only the locking state but also the program counters of all threads.

Let us call lines 12-17 of `InsertPair` its "commit block". Observe that this block is atomic, i.e., any execution in which the actions of this block are interleaved with actions from other threads can be transformed into one in which actions of the commit block

[1] We omit the `Insert` operation to simplify the explanation.

[2] The original implementation of `FindSlot` uses fine-grain locking, and traverses the array using a loop similar to that of `LookUp`. In order to simplify the explanation, we use a version of `FindSlot` that has already been transformed using our proof steps.

Implementation (*Impl*)

```
enum Status = { empty,reserved,full };
record Slot { elt: int, stt: Status };
var M: array[0..N-1] of Slot

LookUp(x:int) returns(r:bool)
  var i: int;
1 for (i := 0; i < N; i++) {
2  lock(M[i]);
3  if (M[i].elt==x && M[i].stt==full){
4   unlock(M[i]);
5   r := true; return;
6  } else unlock(M[i]);
7 }
8 r := false;

atomic FindSlot(x:int) returns (r:int)
1 if (forall 0<=i<N. M[i].stt != empty) {
2  r := -1;
3 } else {
4  assume (0<=r<N && M[r].stt==empty);
5  M[r].stt := reserved;
6 }
```

```
InsertPair(x:int, y:int) returns (r:bool)
   var i,j: int;
 1 i := FindSlot(x);
 2 if (i == -1) {
 3  r := false; return;
 4 }
 5 j := FindSlot(y);
 6 if (j == -1) {
 7  M[i].stt := empty;
 8  r := false; return;
 9 }

10 M[i].elt := x;
11 M[j].elt := y;

12 lock(M[i]);
13 lock(M[j]);
14 M[i].stt := full;
15 M[j].stt := full;
16 unlock(M[i]);
17 unlock(M[j]);
18 r := true;
```

Specification (*Spec*)

```
var S: array [int] of int;
atomic LookUp(x:int) returns (r:bool)
 r := (S[x] > 0);
```

```
atomic InsertPair(x:int, y:int) returns (r:bool)
 if(r) { S[x] := S[x] + 1; S[y] := S[y] + 1; }
```

Figure 1. The concurrent implementation and the sequential specification of multiset

are contiguous. The technique we present allows us to express this fact and use it in a sound manner in a refinement or linearizability proof. Being able to treat the commit block as a single atomic action eliminates all of the potential difficulties outlined above.

Non-fixed commit points. Another issue that complicates the linearizability proof for multiset is that the commit action of `LookUp` is not fixed, but depends on the concurrently executing insertions by other threads. If `LookUp(x)` returns `true` its commit action is at line 3, where it finds out that the slot being visited contains `x` and is valid. When `LookUp(x)` fails, its commit point must be chosen as the first read of a slot it performs or earlier. Intuitively, this is because, in the absence of a `Delete` operation, when a failing `LookUp(x)` starts to execute. However, it is possible that `x` gets inserted into a slot `M[i]`, after `LookUp` visits the ith slot and fails to find `x`, therefore, the commit point cannot be past the first read of a slot. Techniques that depend on the existence of a fixed commit point would be ineffective in such situations [22].

2.2 Proof by reduction and abstraction

In our method the proof is constructed by transforming *Impl* to *Spec*, both shown in Figure 1. This is done through a reduction phase followed by a refinement phase. In the reduction phase, we reduce the bodies of `InsertPair` and `LookUp` to single atomic actions. This phase is guided by a simple hint about the locking discipline (see [6] for details of automating reduction).

In order to handle the non-fixed commit points of `LookUp`, we apply a transformation to separate its succeeding and failing executions. Since each failing iteration of `LookUp` is a left-mover, the failing branch of `LookUp` trivially reduces to a single action. For the successful branch, we apply an abstraction to the failing iterations that makes them also right-movers, and combine the abstracted iterations with the final, successful iteration. This reduces the successful branch into an atomic action. At the end, we obtain `LookUp` as a single atomic action that summarizes both successful and failing executions of the original code.

After transforming `InsertPair` and `LookUp` to single atomic actions, the locking state becomes unnecessary. We use variable hiding to clean up the calls to `lock` and `unlock`. Finally, we arrive at the representation of the multiset in *Spec* in three proof steps. First, we introduce the *Spec* variable `S` to the current version of the program. Then, we add (and prove) following invariant, which links the new variable to the array `M`:

```
S[x]==|{ i | 0<=i<N && M[i].elt==x && M[i].stt==full }|
```

The invariant above allows us to add the assignments `S[x] := S[x] + 1;` and `S[y] := S[y] + 1;` at the end of `InsertPair`. We follow the introduction of `S` with a variable hiding step in which we replace the bodies of `InsertPair` and `LookUp` with the corresponding bodies in *Spec* (Figure 1). Our soundness theorems given in Section 5 guarantee that transforming *Impl* to *Spec* using our rules implies the linearizability of *Impl*.

What is noteworthy about the proof we outlined is that it handles two separate concerns in separate proof steps: 1) concurrency control using locking and the `stt` field, and 2) relating the array-based representation of *Impl* to the representation in *Spec*. This example does not illustrate the use of variable hiding to eliminate superficial conflicts. In Section 6 we provide an example that does.

3. Concurrent programs: syntax and semantics

Program. A program P is a tuple $P = \langle Global_P, Proc_P \rangle$. $Global_P$ is the set of uniquely-named global variables. $Proc_P$ is a set of procedures. A procedure is a tuple $\langle \rho, local_\rho, body_\rho \rangle$, where ρ is the *name*, $local_\rho$ is the set of *local variables*, and $body_\rho$ is the *body* of the procedure.

We distinguish the input variables $\overrightarrow{in}_\rho \subseteq local_\rho$ and the output variables $\overrightarrow{out}_\rho \subseteq local_\rho$. The tuple $\langle \rho, \overrightarrow{in}_\rho, \overrightarrow{out}_\rho \rangle$ is called the *signature* of the procedure. The signatures of the procedures in $Proc$ form the signature of the program, denoted $\mathsf{Sig}(P)$. We employ the convention that the variables in $\overrightarrow{in}_\rho$ and $\overrightarrow{out}_\rho$ are read-only and write-only, respectively, while the rest of the variables in $local_\rho$ can be both read and updated.

We use Var_P to denote $Global_P \cup \bigcup_{\rho \in Proc} local_\rho$. We assume that each local variable is used in a unique procedure. Var'_P consisting of the primed version of each variable in Var_P. We omit the

subscripts when the program and the procedure are clear from the context.

Execution model. Let Tid be the set of all thread identifiers. For simplicity of presentation, we assume that procedure calls are inlined properly, assuming no recursion in the call chain. In general, our method applies to the inter-procedural case allowing recursion [5].

Without loss of generality, each thread calls one procedure ρ from $Proc$, and terminates when ρ returns. Statements of the procedures may refer to the current thread id through the special variable $tid \in Global$, whose domain is Tid.

Syntax. We assume that each atomic statement α, which we call an *(atomic) action*, is in the form: $\mathsf{assert}\ a; p$. Let ρ be the procedure whose body contains α, and $V = Global \cup \mathit{local}_\rho$. The *assert predicate* a be over only unprimed variables from V. The *transition predicate* p is over both primed and unprimed variables in $V \cup V'$. For any action α, let ϕ_α and τ_α denote its assert and transition predicates. For instance, $\phi_\alpha = a$ and $\tau_\alpha = p$, for α given above.

We use sequential composition (;), choice ($\Box$) and loop ($^\circlearrowleft$) operators to form *compound statements*. We also define the *nullary action* stop, which appears only at runtime and intuitively marks the end of fully executing a statement.

Program states. A program state s is a pair consisting of

- a *variable valuation* σ_s that maps a thread id and a variable to a value, such that $\sigma_s(t, g) = \sigma_s(u, g)$ for all states s and thread id's t, u, whenever g is a global variable.
- a *code map* ϵ_s that keeps track of a (compound) statement for each thread, such that $\epsilon_s(t) = c$ means that at program state s, the remaining part of the program to be executed by thread t is given by c.

A program state s is called *initial* if $\forall t \in Tid.\ \exists \rho \in Proc.\ \epsilon_s(t) = \mathit{body}_\rho$, i.e. every thread is about to call a procedure. State s is called *final* if $\epsilon_s(t) = \mathsf{stop}$, for all $t \in Tid$. We write $Initial(s)$ (resp. $Final(s)$) to denote that s is an initial (resp. final) state.

Let $\sigma_s|_V$ denote the projection of valuation σ_s on $V \subseteq Var$. Define $s|_V$ to be the program state $(\sigma_s|_V, \epsilon_s)$. This definition also pointwise applies to collections of states.

Predicates over program variables. For an assert predicate x, let $x[t]$ denote the predicate in which all free occurrences of tid is replaced with t. We say that a program state s satisfies $x[t]$, denoted as $s \vDash x[t]$ or as $x[t](s)$, if $x[t]$ evaluates to true when all free occurrences of each unprimed variable v is replaced with $\sigma_s(t, v)$. An assert predicate is called a *state predicate* if it does not contain any free occurrence of tid.

Similarly, the pair of program states (s_1, s_2) satisfies a transition predicate $p[t]$, denoted as $(s_1, s_2) \vDash p[t]$ or as $p[t](s_1, s_2)$, if $p[t]$ evaluates to true when each unprimed variable v (resp. v') is replaced with $\sigma_{s_1}(t, v)$ (resp. $\sigma_{s_2}(t, v)$).

We define $\mathsf{fv}(p)$ to be the set of free variables in the (state or transition) predicate p.

3.1 Operational semantics

Configurations. The evaluation of a full statement is given in terms of the *silent transformation relation*, $\hookrightarrow$, whose definition is given in Fig. 2. Intuitively, if we imagine the execution of a full statement represented as a flowchart with an explicit control pointer denoting what to execute next, the silent transformation relation corresponds to advancing the control pointer over the flowchart not modifying any program variable's value. When this imaginary control pointer selects a branch, it is represented by the label λ which is called the *invisible transition*. Otherwise, the label is the content of the box over which the control pointer passes.

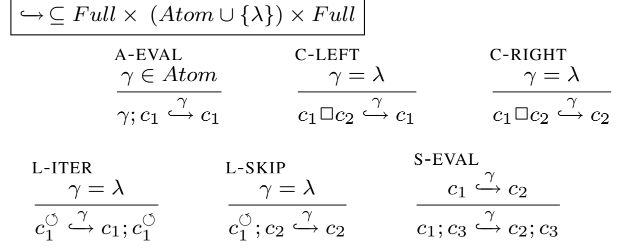

Figure 2. Obtaining all possible subactions of a given full action via the silent transformation relation, $\hookrightarrow$.

For full actions c and d, and a string $\overline{\gamma} = \gamma_1 \dots \gamma_n$ over $Atom \cup \{\lambda\}$, we let $c \overset{\overline{\gamma}}{\hookrightarrow} d$ denote a sequence of silent transformations

$$c = c_0 \overset{\gamma_1}{\hookrightarrow} c_1 \dots \overset{\gamma_n}{\hookrightarrow} c_n = d$$

A program state s' is in $\mathsf{conf}(s)$, the *configurations* reachable from program state s, if, for all t, there exists some string $\overline{\gamma}_t$ such that $\epsilon_s(t) \overset{\overline{\gamma}_t}{\hookrightarrow} \epsilon_{s'}(t)$. Intuitively, s' is a configuration of s if s' can be obtained by moving forward the control pointer of each thread's program an arbitrary number of, possibly 0, steps.

Let s and s' be program states, t be a thread id. Then, s' is called a (t, α)-*successor* of s (or s, a (t, α)-*predecessor of* s', if the following conditions hold:

- $\epsilon_s(t) \overset{\lambda^k \alpha}{\hookrightarrow} \epsilon_{s'}(t)$, for some $k \geq 0$.
- for all $u \neq t$, $\epsilon_{s'}(u) = \epsilon_s(u)$,

Intuitively, s' is a (t, α)-successor of s if at s thread t has α as a possible next action and s' is the same as s except the control flow at t skips over α. For any thread t and $\gamma \in Atom$, (t, γ) is called a *transition label*.

Execution semantics. We assume a sequentially-consistent memory model. We say $s \xrightarrow{(t,\alpha)} s'$ holds when t can execute α next (in which case s' is a (t, α) successor of s), all other threads do not update their control flow, all local variables of other threads remain the same, the global variables and local variables of t are updated so that the transition predicate of α is satisfied. Formally, $s \xrightarrow{(t,\alpha)} s'$ if $(s, s') \vDash \tau_\alpha[t]$ and for all $u \neq t$ and for any local variable x, $\sigma_s(u, x) = \sigma_{s'}(u, x)$.

Run. A run r of the program is a sequence of state transitions:

$$r = r_1 \xrightarrow{(t_1,\alpha_1)} r_2 \xrightarrow{(t_2,\alpha_2)} \cdots \xrightarrow{(t_{n-1},\alpha_{n-1})} r_n$$

For the definitions that follow, we fix the run r above. Let $Tid(r)$ denote the set of threads occurring in r. Let r_i denote the ith program state, and $r(i)$, the ith transition label (t_i, α_i) in r. For a state predicate ϕ, we say that r is a run of P from ϕ if $Initial(r_1)$ and $r_1 \vDash \phi$.

The run is *maximal* if r_n cannot make any transition. Henceforth, we will only consider maximal runs.

Trace. A *trace* is a sequence of transition labels, $\mathbf{l} = l_1 \dots l_k$. The trace moves a state s_1 to s_{k+1}, written $s_1 \xrightarrow{\mathbf{l}} s_{k+1}$, if there is a run r of P over $\mathbf{l}$, such that $r_j = s_j$, for all $1 \leq j \leq k+1$ and $r_i \xrightarrow{l_i} r_{i+1}$.

Violation-freedom. A run r of P from ϕ is called a *violation* if $\neg\phi_\alpha[t](r_k)$ evaluates to true for some $(t, \alpha) \in \mathsf{next}(r_k)$. Intuitively, a violation is a run of P that starts from an initial program

state s_1 and reaches a program state s_k which violates the assert predicate, ϕ_α, of an action α which thread t can execute at state s_k. A run is said to be *successful* if it is not a violation. We indicate a successful run as $s_1 \xrightarrow{1} s_2$ and a violation as $s_1 \xrightarrow{1} \mathsf{error}$.

4. Program transformations

In this section, we formalize our notion of proof and introduce the rules for the proof calculus. A *proof state* is the pair $(P, \mathcal{I})$, where P is a program, and $\mathcal{I}$ is a state predicate, called the *inductive invariant* of the program. We require that for every proof state $(P, \mathcal{I})$, all the atomic actions of P preserve $\mathcal{I}$. An atomic action α preserves $\mathcal{I}$, written $\alpha \leftrightarrows \mathcal{I}$, if $s_1 \xrightarrow{(t,\alpha)} s_2$ and $s_1 \vDash \mathcal{I}$ imply $s_2 \vDash \mathcal{I}$.

A proof consists of rewriting the input program, denoted P_1, iteratively so that, in the limit, one arrives at a program, denoted P_n, that can be verified by sequential reasoning methods. Formally, the proof is expressed as $(P_1, \mathsf{true}) \dashrightarrow (P_2, \mathcal{I}_2) \dashrightarrow \cdots \dashrightarrow (P_n, \mathcal{I}_n)$ Each proof step is governed by a proof rule, which we present below.

The following proof rule states the general form of updating $\mathcal{I}$, replacing it with a stronger invariant.

RULE 1 (Invariant). *Replace invariant $\mathcal{I}_1$ with $\mathcal{I}_2$ if $\alpha \leftrightarrows \mathcal{I}_2$ for all the actions α in P, and $\mathcal{I}_2 \Rightarrow \mathcal{I}_1$.*

The basic idea in reduction and abstraction is to replace an action with another action that simulates the former.

DEFINITION 1 (Simulation). *Let α, β be actions, t be an arbitrary thread id. We say β simulates α at proof state $(P, \mathcal{I})$, written $(P, \mathcal{I}) \vdash \alpha \preceq \beta$, if both of the following hold:*

$$\begin{array}{llcl} \mathbf{S1}. & (\mathcal{I} \wedge \neg\phi_\alpha) & \Rightarrow & \neg\phi_\beta \\ \mathbf{S2}. & (\mathcal{I} \wedge \tau_\alpha) & \Rightarrow & (\neg\phi_\beta \vee \tau_\beta) \end{array}$$

Intuitively, **S1** states that if there is a violation with α, there has to be a violation with β substituted in place of α. **S2** states that for each violation-free run, replacing α with β results in either a violation, or a violation-free run with the same end state.

4.1 Reduction

Reduction, due to Lipton [16], creates coarse-grained atomic statements by combining fine-grained actions. An action α can be combined with another action if α is a certain kind of mover. A mover is an action that can commute over actions of other threads in any run. We write $(P, \mathcal{I}) \vdash \alpha : m$ to indicate that α is $m-$mover in the proof state $(P, \mathcal{I})$, where $m \in \{\mathbb{L}, \mathbb{R}\}$.

We decide that an action α is a mover by statically checking a simulation relation, that states that commuting α with every β can lead to the same state or goes wrong. An assert predicate x is *p-stable*, if $\forall s, s'.x(s) \wedge p(s, s') \Rightarrow x(s')$.

Let $\mathsf{wp}(p, x)$, the *weakest (liberal) pre-condition* of predicate x for transition predicate p, stand for all states which cannot reach a state where x evaluates to false after executing p. Formally, $\mathsf{wp}(p, x) = \{s \mid \forall s'. p(s, s') \Rightarrow x(s')\}$. For two transition predicates p and q, define their composition $p \cdot q$, as the transition predicate $p \cdot q = \{(s_1, s_2) \mid \exists s_3. p(s_1, s_3) \wedge q(s_3, s_2)\}$. The operator $[\![\ ]\!]$ expresses the result of combining two actions to one atomic action.[3]

$$\begin{array}{rcl} [\![\alpha; \beta]\!] & = & \mathsf{assert}\,(\phi_\alpha \wedge \mathsf{wp}(\tau_\alpha, \phi_\beta)); (\tau_\alpha \cdot \tau_\beta) \\ [\![\alpha \Box \beta]\!] & = & \mathsf{assert}\,(\phi_\alpha \wedge \phi_\beta); (\tau_\alpha \vee \tau_\beta) \end{array}$$

[3] We assume that a transition predicate $\tau_\alpha[t]$ can only change the variables in the scope of t and that if t and u are running the same procedure, local variables are suitably renamed to prevent false conflicts.

DEFINITION 2 (Left-mover). *Action α is a left-mover in proof state $(P, \mathcal{I})$, denoted $(P, \mathcal{I}) \vdash \alpha : \mathbb{L}$, if the following holds for every action β in P and every pair of distinct thread ids t and u: $(P, \mathcal{I}) \vdash [\![\beta[u]\ ;\ \alpha[t]]\!] \preceq [\![\alpha[t]\ ;\ \beta[u]]\!]$.*

DEFINITION 3 (Right-mover). *Action α is a right-mover in proof state $(P, \mathcal{I})$, denoted $(P, \mathcal{I}) \vdash \alpha : \mathbb{R}$, if, for every action β in P, and every pair of distinct thread ids t and u: $(P, \mathcal{I}) \vdash [\![\alpha[t]\ ;\ \beta[u]]\!] \preceq [\![\beta[u]\ ;\ \alpha[t]]\!]$ and $\phi_\beta[u]$ is $\tau_\alpha[t]$-stable.*

The reduction rules below define the conditions under which non-atomic statements are transformed to atomic actions. We omit the rules about procedure calls and parallel composition which are similar to those of [5].

RULE 2 (Reduce-Sequential). *Replace occurrences of $\alpha\ ;\ \gamma$ with $[\![\alpha\ ;\ \gamma]\!]$ if either $(P, \mathcal{I}) \vdash \alpha : \mathbb{R}$ or $(P, \mathcal{I}) \vdash \gamma : \mathbb{L}$.*

RULE 3 (Reduce-Choice). *Replace occurrences of $\alpha \Box \gamma$ with $[\![\alpha \Box \gamma]\!]$.*

RULE 4 (Reduce-Loop). *Replace occurrences of $\alpha^{\circlearrowleft}$ with β if the following hold:*

$$\begin{array}{ll} \mathbf{L1}. & (P, \mathcal{I}) \vdash \alpha : m \text{ s.t. } m \in \{\mathbb{R}, \mathbb{L}\} \\ \mathbf{L2}. & \beta \leftrightarrows \mathcal{I} \\ \mathbf{L3}. & \phi_\beta \Rightarrow \tau_\beta[\mathit{Var}/\mathit{Var}'] \\ \mathbf{L4}. & (P, \mathcal{I}) \vdash [\![\beta\ ;\ \alpha]\!] \preceq \alpha \end{array}$$

4.2 Abstraction

The purpose of the abstraction rule is to replace an action with another action An abstraction step consists of replacing an action α with another action β, which in principle leads to less interference with other actions.

RULE 5 (Abstraction). *Replace the action α with action β if $\beta \leftrightarrows \mathcal{I}$ and $(P, \mathcal{I}) \vdash \alpha \preceq \beta$.*

This rule is usually applied for an action $\mathsf{assert}\ a; p$ by replacing it with 1) $\mathsf{assert}\ b; p$ such that $b \Rightarrow a$ or 2) with $\mathsf{assert}\ a; q$ such that $p \Rightarrow q$. While the former corresponds to adding extra assertions to the action, the latter adds more (non-deterministic) transitions.

4.3 Variable introduction and hiding

Intuitively, variable introduction rewrites some actions in the program so that these can refer to a new (*history*) variable. Variable hiding is the dual of variable introduction; each action is rewritten so that it does no longer refer to the hidden variable. Hiding a variable also requires quantifying out the variable in the invariant.

In order to ensure soundness, in both cases, we need a relation between actions over different sets of variables. For this, we extend our simulation relation ($\preceq$) for each rule. In addition, we require that the input and output variables of the procedures ($\overrightarrow{in}_\rho, \overrightarrow{out}_\rho$) are fixed during the proof; the rules below are not applicable to these variables.

RULE 6 (Add-Variable). *Add the new variable v to Var_P, and replace every action α with β whenever $(P, \mathcal{I}) \vdash \alpha \preceq_{+v} \beta$, which holds if the following are both valid:*

$$\begin{array}{llcl} \mathbf{A1}. & (\mathcal{I} \wedge \neg\phi_\alpha) & \Rightarrow & (\forall v.\ \neg\phi_\beta) \\ \mathbf{A2}. & (\mathcal{I} \wedge \tau_\alpha) & \Rightarrow & (\forall v.\ \neg\phi_\beta \vee (\exists v'.\ \tau_\beta)) \end{array}$$

RULE 7 (Hide-Variable). *Remove the existing variable v from the program, and replace the invariant $\mathcal{I}$ with $\exists v.\ \mathcal{I}$. Replace every action α with β whenever $(P, \mathcal{I}) \vdash \alpha \preceq_{-v} \beta$, which holds if the following are both valid:*

$$\begin{array}{llcl} \mathbf{H1}. & (\exists v.\ \mathcal{I} \wedge \neg\phi_\alpha) & \Rightarrow & \neg\phi_\beta \\ \mathbf{H2}. & (\exists v, v'.\ \mathcal{I} \wedge \tau_\alpha) & \Rightarrow & (\neg\phi_\beta \vee \tau_\beta) \end{array}$$

Fix a thread t and a state s. In both of the rules, the first condition (**A1**, **H1**) states that violations are preserved. The second condition (**A2**, **H2**) states that transitions (over the common variables of α and β) are either preserved or additional violations are introduced.

5. Soundness theorems

Given a proof $(P_1, \mathcal{I}_1) \dashrightarrow \cdots \dashrightarrow (P_n, \mathcal{I}_n)$, we now provide the soundness theorems. Each theorem relates P_n to P_1, providing a soundness guarantee for a particular use of our method.

5.1 Proving assertions

The first theorem is an extension of the main soundness theorem in [5]. Intuitively, the theorem states that proof steps preserve violations, and initial-final state pairs when the output program is good from the final invariant.

Good and Bad. In the following, we define $Good(P, \mathcal{I})$ as the set of pre- and post-state pairs associated with succeeding (maximal) runs of program P from states satisfying $\mathcal{I}$. $Bad(P, \mathcal{I})$ is the set of pre-states associated with violations. Formally,

$$Good(P,\mathcal{I}) = \{(s_1, s_2) \mid Initial(s_1),\ s_1 \vDash \mathcal{I},\ \exists \mathbf{l}.\ s_1 \xrightarrow{1} s_2,\ Final(s_2)\}$$
$$Bad(P,\mathcal{I}) \ = \{s_1 \mid Initial(s_1),\ s_1 \vDash \mathcal{I},\ \exists \mathbf{l}.\ s_1 \xrightarrow{1} \mathsf{error}\}$$

P is said to be *good* from $\mathcal{I}$ if $Bad(P, \mathcal{I}) = \emptyset$; it is called *bad* from $\mathcal{I}$, otherwise.

THEOREM 1. *Let* $(P_1, \mathcal{I}_1) \dashrightarrow \cdots \dashrightarrow (P_n, \mathcal{I}_n)$ *be a sequence of proof steps. Let* $V = Var_{P_1} \cap Var_{P_n}$ *and* $X = (Var_{P_1} \cup Var_{P_n}) \setminus \mathrm{V}$. *The following hold:*

C1. $Bad|_V(P_1, \exists X.\, \mathcal{I}_n) \subseteq Bad|_V(P_n, \exists X.\, \mathcal{I}_n)$
C2. $\forall (s_1, s_n) \in Good|_V(P_1, \exists X.\, \mathcal{I}_n)$:
  a. $s_1 \in Bad|_V(P_n, \exists X.\, \mathcal{I}_n)$ *or*
  b. $(s_1, s_n) \in Good|_V(P_n, \exists X.\, \mathcal{I}_n)$

Note that, since the input and output variables of procedures are fixed during the proof, so the set V above will always be nonempty. A corollary of the above theorem is that, if P_n is good from $\mathcal{I}_n$, then P_1 is good from $\mathcal{I}_n$. This means that, one can prove the assertions in P_1 by gradually obtaining programs with coarser-grained concurrency using our proof rules. At any point during this process, the decreased level of granularity will simplify the checking of assertions. We give the proof of the theorem in Appendix Section B.

5.2 Proving linearizability

In this section, we establish a link between P_1 and P_n in the context of proving linearizability. For this, we first define *behavioral simulation*, a special kind of simulation that relates two programs through their observable behaviors over procedure input and output values.

Behavioral simulation. Let $r = s_1 \xrightarrow{1} s_n$ be a (maximal) run of the program. Let ρ be the procedure executed by t. We call the tuple $(t, \rho, \sigma_{s_1}(t, \overrightarrow{in}_\rho), \sigma_{s_n}(t, \overrightarrow{out}_\rho))$ the behavior of t in r and denote it by $\mathsf{beh}(r, t)$. The behavior includes the name of the procedure called by t, along with the values of the input and the output variables of the procedure[4]. We write $\mathsf{Beh}(r)$ to denote $\{\mathsf{beh}(r, t) \mid t \in Tid(r)\}$.

We define $\mathsf{fst}(r, t)$ and $\mathsf{lst}(r, t)$ be the indices of first and the last actions of t in r. Formally, with $L = \{i \mid r(i) = (t, \alpha)\}$, $\mathsf{fst}(r, t) = \min(L)$ and $\mathsf{lst}(r, t) = \max(L)$. Let $\ll_r$ be a partial order over $Tid(r)$ ordering threads that do not execute concurrently: $t \ll_r u$ if $\mathsf{lst}(r, t) < \mathsf{fst}(r, u)$.

DEFINITION 4. *Let P and P' be two programs with* $\mathsf{Sig}(P) = \mathsf{Sig}(P')$*, and let $\mathcal{I}$ be a state predicate. Let* $X_1 = \mathsf{fv}(\mathcal{I}) \setminus Var_P$ *and* $X_2 = \mathsf{fv}(\mathcal{I}) \setminus Var_{P'}$. *$P'$ behaviorally-simulates P from $\mathcal{I}$, denoted $P \lhd_\mathcal{I} P'$ if for each maximal run r of program P from $\exists X_1.\mathcal{I}$, there exists a maximal run r' of P' from $\exists X_2.\mathcal{I}$ such that 1)* $\mathsf{Beh}(r) = \mathsf{Beh}(r')$ *and 2)* $\ll_r \subseteq \ll_{r'}$

Linearizability. We now formalize the generic notion of linearizability in [10]. In this notion a run of the program is described by a sequence of meta-actions invoke and response. The sequence of these meta-actions is called the history of the run. The history of the run r is denoted $\mathbf{H}_r$, and $\mathbf{H}(i)$ denotes the ith meta-action in $\mathbf{H}$.

$Tid(\mathbf{H})$ denotes the set of thread identifiers in $\mathbf{H}$. Let $\mathbf{H}|t$ be the subhistory of $\mathbf{H}$ projected on the thread t.

An invoke action $\mathsf{inv}\langle \rho, xs, t\rangle$ describes a call to a procedure ρ with arguments xs by thread t. A response action $\mathsf{res}\langle \rho, ys, t\rangle$ describes a return from the procedure ρ with return values ys in thread t. We will often omit the elements of the meta-actions and refer to them using the dotted notation (e.g., $\mathsf{inv}.\rho$).

We say that $(\mathsf{inv}, \mathsf{res})$ is a *pair of matching invoke and response* in $\mathbf{H}$, if $\mathbf{H}|t(i) = \mathsf{inv}$ and $\mathbf{H}|t(i+1) = \mathsf{res}$ for some thread t and index i. We say that a history $\mathbf{H}$ is *sequential*, if for each matching pair $(\mathsf{inv}, \mathsf{res})$ in $\mathbf{H}$, $\mathbf{H}(i) = \mathsf{inv}$ and $\mathbf{H}(i+1) = \mathsf{res}$ for some index i.

We say $\mathbf{H}$ and $\mathbf{H}'$ are *equivalent*, denoted by $\mathbf{H} \cong \mathbf{H}'$, if $Tid(\mathbf{H}) = Tid(\mathbf{H}')$ and $\forall t \in Tid(\mathbf{H})$. $\mathbf{H}|t = \mathbf{H}'|t$.

Given a history $\mathbf{H}$, we define *response-invoke preservation relation* $\preceq_\mathbf{H}$ as follows:

> Two matching invoke-response pairs $(\mathsf{inv}, \mathsf{res})$ and $(\mathsf{inv}', \mathsf{res}')$ in $\mathbf{H}$ are ordered, denoted, $(\mathsf{inv}, \mathsf{res}) \preceq_\mathbf{H} (\mathsf{inv}', \mathsf{res}')$, if $\mathbf{H}(i) = \mathsf{res}$ and $\mathbf{H}(j) = \mathsf{inv}'$ for some $i < j$.

A history $\mathbf{H}$ is linearizable to a sequential history $\mathbf{H}'$ if $\mathbf{H} \cong \mathbf{H}'$ and $\preceq_\mathbf{H} \subseteq \preceq_{\mathbf{H}'}$.

DEFINITION 5 (Linearizability). *Let P' be an atomic program and $\mathcal{I}$ be an assert predicate. A program P is* linearizable to P' from $\mathcal{I}$ *if every history of P from $\mathcal{I}$ is linearizable to some sequential history of P' from $\mathcal{I}$.*

The following theorem connects behavioral simulation to the generic notion of linearizability. We say *P is linearizable to P' from $\mathcal{I}$* to restrict the definition of linearizability to runs of P and P' from $\mathcal{I}$. A program P is called an *atomic program* if for every $\rho \in Proc_P$, body_ρ is an atomic action.

THEOREM 2. *Let P' be an atomic program that is good from $\mathcal{I}$. A program P is* linearizable to P' *from $\mathcal{I}$ iff $P \lhd_\mathcal{I} P'$.*

PROOF.

CASE: $\rightarrow$
 ASSUME: P is linearizable to P' from $\mathcal{I}$
 PROVE: $P \lhd_\mathcal{I} P'$
  $\langle 3\rangle 1.$ Take a run r of P from $\mathcal{I}$. Consider its history $\mathbf{H}_r$. By assumption, there exists a sequential history $\mathbf{H}'$ of P' from $\mathcal{I}$ such that $\mathbf{H}$ is linearizable to $\mathbf{H}'$. Then there exists a run r' of P' from which $\mathbf{H}'$ is obtained from, so that $Tid(\mathbf{H}) = Tid(\mathbf{H}') = Tid(r) = Tid(r')$. In addition, $\mathsf{Beh}(r') = \mathsf{Beh}(r)$ since the input and output values from $\mathbf{H}$ and $\mathbf{H}'$ are equivalent. In addition, $\preceq_\mathbf{H} \subseteq \preceq_{\mathbf{H}'}$ implies $\ll_r \subseteq \ll_{r'}$. Thus for every r, we can find a run r' that will allow us to show that $P \lhd_\mathcal{I} P'$.

[4] Notice that the first and the last states of the run provide us the values of $\overrightarrow{in}_\rho$ and $\overrightarrow{out}_\rho$, respectively.

$\langle 2\rangle 1.$ Q.E.D.

CASE: $\leftarrow$

PROVE: $P \lhd_{\mathcal{I}} P'$

ASSUME: P is linearizable to P' from $\mathcal{I}$

$\langle 3\rangle 1.$ Let $\mathbf{H}$ be history of program P. We have a run r of P from which we obtain $\mathbf{H}$. Since $P \lhd_{\mathcal{I}} P'$, there exits a run r' of P' such that $\mathsf{Beh}(r) = \mathsf{Beh}(r')$ and $\ll_r \subseteq \ll_{r'}$. We first construct sequential history $\mathbf{H}'$ of r' of atomic program P' as described in the definition of a history by means of invocation and return of procedures in r'. Since $\mathsf{Beh}(r) = \mathsf{Beh}(r')$ we know that $Tid(r) = Tid(r')$ and hence $Tid(\mathbf{H}) = Tid(\mathbf{H}')$ and $\forall t \in Tid(r)$, $\mathsf{beh}(r,t) = \mathsf{beh}(r',t)$. Thus, $\forall t \in Tid(\mathbf{H})$ $\mathbf{H}|t = \mathbf{H}'|t$. That is, $\mathbf{H} \cong \mathbf{H}'$. We also have $\ll_r \subseteq \ll_{r'}$ which implies $\preceq_{\mathbf{H}} \subseteq \preceq_{\mathbf{H}'}$ since we do not change order of actions when deriving the history of a run. Hence, $\mathbf{H}$ is linearizable to $\mathbf{H}'$, and P is linearizable to P'.

$\langle 2\rangle 1.$ Q.E.D.

$\langle 1\rangle 1.$ Q.E.D.

The following theorem states that each good program reached during the proof behaviorally simulates the initial program.

THEOREM 3 (Soundness). *Let $(P_1, \mathcal{I}_1) \dashrightarrow \cdots \dashrightarrow (P_n, \mathcal{I}_n)$ be a sequence of proof steps such that P_n is good from $\mathcal{I}_n$. Then forall $1 \le i \le n$, $P_1 \lhd_{\mathcal{I}_n} P_i$ holds.*

We give the proof of the theorem in Appendix Section A.

Theorems 2 and 3 provide two options for proving linearizability of P_1 to the intended specification from $\mathcal{I}$, represented by an atomic program P_n. First, one can complement another proof method with ours, by first performing the proof $(P_1, \mathsf{true}) \dashrightarrow \cdots \dashrightarrow (P_k, \mathcal{I})$, and then applying her method to prove that P_k is linearizable to P_n. Once the proof passes, this implies that P_1 is also linearizable to P_n, since our transformations preserve all the behaviors of the program relevant to linearizability. Alternatively, s/he can keep transforming $(P_k, \mathcal{I})$ up to $(P_n, \mathcal{I})$, and complete the full proof of linearizability in our system. Note that, for the theorems to ensure soundness in these cases, s/he must also prove that P_k (resp. P_n) is good from $\mathcal{I}$. The latter is formalized by the following.

COROLLARY 4. *Let $(P_1, \mathsf{true}) \dashrightarrow \cdots \dashrightarrow (P_n, \mathcal{I})$ be a sequence of proof steps, such that P_n is an atomic program that is good from $\mathcal{I}$. Then, P_1 is linearizable to P_n from $\mathcal{I}$.*

6. Implementation and experience

We implemented our proof method in the QED verifier. QED accepts as input a multithreaded program written in an extension of the Boogie programming language and a proof script. All the transformations are applied automatically, and when necessary, the preconditions of the transformations are checked, by generating verification conditions and feeding them to the Z3 SMT solver. Using QED, we mechanically proved the linearizability of the following programs:

- Lock-coupling linked list [22]
- Treiber's non-blocking stack [20]
- Non-blocking and two-lock queues [17]
- Non-blocking mutex lock implementation adapted from [14]

For each data structure, we chose a generic specification as the target of the proof, and were able to transform the program to the specification program through few reduction and refinement phases. The QED tool and the proof scripts of the above programs are available at *http://qed.codeplex.com*.

In the rest of the section, we overview the proof of the non-blocking queue, and describe how coupling variable introduction and hiding helps us to cope with superficial conflicts. This is an important limitation for reduction, and interestingly, our standard notion of abstraction on the existing variables (Section 4.2) does not help in this situation. Our solution to eliminating the conflict is to hide the variables on which the conflict happens; but, differently from the standard abstraction, introducing new variables, which will carry enough (semantic) information from the hidden variables and will not cause conflicts.

6.1 Non-blocking queue

Figure 3 shows the version of the non-blocking queue [17] after applying a reduction phase on the original implementation. Atomic action `Do_Dequeue` removes an element from the queue, and `Do_Enqueue` appends a new element to the queue. The implementation is lazy in that `Do_Enqueue` does not update the `Tail` variable after adding the new node. As a result, at any time `Tail` may point to any node between `Head` and `null`. The actions labeled `Move_Tail` and `Update_Tail` try to move the `Tail` towards the end of the list. This resembles relaxed balancing in concurrent implementation of tree-like data structures, in which restructuring the data structure is separated from actual operations, and delayed.

The predicate `Reach(next,k,l,m)` expresses that, from node `k`, following zero or more `next` pointers, we first reach `l` and then `m` [15]. The `Reach` predicate gives us the ability to do simple abstractions on actions accessing the list nodes. For example, a former abstraction step in the reduction phase replaces the action `n := tail.next` with the action `havoc n; assume Reach(next,tail,n,n)`; while the former is not mover, the latter is.

In order to apply reduction, the only option is to show that `Move_Tail` is a right-mover, since `Do_Enqueue` and `Do_Dequeue` perform the actual operations, thus are not movers. `Move_Tail` conflicts with `Do_Enqueue` and `Do_Dequeue` on `Tail`. Notice that `Move_Tail` performs an internal operation that does not affect the semantics of the queue. Thus, these conflicts are superficial. Havocing `Tail` in the conflicting actions, or hiding `Tail` are a valid proof steps, and would make reduction pass. However, the resulting code would perform incorrect operation.

We eliminate the conflict by coupling the hiding of `Tail` with introducing the history variable `_Tail` of the same type. Differently from `Tail`, `_Tail` always points to the end of the queue. We then associate the existing variables with the new variable `_Tail` by the following invariant.

```
invariant Reach(next, Head, Tail, Tail)
          && Reach(next, Tail, _Tail, null)
          && (_Tail != null) && (_Tail.next == null)
```

In order to satisfy the invariant, we add to the end of `Do_Enqueue` the assignment `_Tail := _Tail.next`. Once there is `_Tail` to keep track of the end of the list, we are ready to hide `Tail`. This is done by replacing the actions in the program with actions that do not refer to `Tail`, but now uses `_Tail` to access the end of the linked list. Figure 4 shows the version of the program after hiding `Tail`. Notice that the new form of `Move_Tail` does not perform any semantic operation in the new program, and does not conflict with other actions. In addition, the actions `Do_Enqueue` and `Do_Dequeue` now use `_Tail` to correctly perform their operations.

The hiding step also includes existentially quantifying `Tail` in the invariant given above. This produces the following invariant for the new program.

```
invariant Reach(next, Head, _Tail, null)
          && (_Tail != null) && (_Tail.next == null)
```

Implementation (*Impl*)

```
record Node { data: int; next: Node; }
var Head, Tail: Node;

Dequeue() returns (x: int)
 var tail: Node;
 while(true) {
Move_Tail: atomic {
    havoc tail;
    assume Reach(next, Tail, tail, null)
           && tail != null;
    Tail := tail;
   }
 } // end while
Do_Dequeue: atomic {
  if (Head.next == null) {
   x := null;
  } else {
   assume (Head != Tail);
   Head := Head.next;  x := Head.data;
  }
 }
```

```
Enqueue(x: int)
 var node, tail: Node;
 atomic {
  node := new Node(x);
  node.next := null;
 }
 while(true) {
Move_Tail:atomic {
    havoc tail;
    assume Reach(next, Tail, tail, null)
           && tail != null;
    Tail := tail;
   }
 } // end while
Do_Enqueue: atomic {
  assume (Tail.next == null);
  Tail.next := node;  tail := Tail;
 }
Update_Tail: atomic {
  if (Tail == tail) Tail := node;
 }
```

Specification (*Spec*)

```
atomic Dequeue() returns (x: int)
  if (Head.next == null) {
   return null;
  } else {
   Head := Head.next; x := Head.data;
  }
```

```
atomic Enqueue(x: int)
  node := new Node(x);
  node.next := null;
  _Tail.next := node;
  _Tail := _Tail.next;
```

Figure 3. The reduced implementation of the non-blocking queue and its specification

```
record Node { data: int; next: Node; }
var Head, _Tail: Node;

procedure Dequeue()
 var tail: Node;

 while(true) {
Move_Tail: atomic { havoc tail; }
 } // end while

Do_Dequeue: atomic {
  if (Head.next == null) {
   x := null;
  } else {
   Head := Head.next;  x := Head.data;
  }
 }
```

```
procedure Enqueue(x: int)
 var node, tail: Node;
 atomic {
  node := new Node(x);
  node.next := null;
 }

 while(true) {
Move_Tail: atomic { havoc tail; }
 } // end while

Do_Enqueue: atomic {
   _Tail.next := node;
   _Tail := _Tail.next;
   havoc tail;
 }
Update_Tail: atomic { assume true; }
```

Figure 4. The version of the non-blocking queue after hiding `Tail`

We proceed with a reduction phase that combines the blocks into a single action for each operation. The combined operations, together with the above invariant (for simplicity, we omit parts of the representation invariant), give the correct behavior of a sequential queue implementation. Corollary 4 ensures that the original implementation in [17] is linearizable to this final program from the invariant. Note that it also possible to continue the proof with an extra refinement phase to prove the linearizability to a more generic specification of the queue.

6.2 The mutex lock example

Our mutex lock implementation is based on the writer part of the readers/writer lock in [14]. Figure 5 shows the version of the implementation after a reduction phase. Similarly to the queue example, we will explain the application of the refinement phase to transform that version to the specification given in the same figure.

In the example, locking requests are kept as nodes in a linked list. At the head of the list is the node holding the lock, and the rest belong to the threads waiting for the lock to become free. When the head node releases the lock, it signals its next node to allow it to grab the lock. The `spin` field is used for this wait/signal protocol.

When the list is empty, the variable `L` is `null`, otherwise `L` points to the tail of the list. `Lock` creates a node named `I` and appends it to the tail of the list. If the list is empty, the action `Create_Node` completes the acquision of the lock. Otherwise, the action `Wait_Prev` waits, on its for the previous node to signal it.

The created node `I` is returned to the caller, which later passes it to the `Unlock` procedure. Action `Check_Empty` of `Unlock` completes the release if `I=L`, i.e. `I` is the only node in the list. Otherwise, `Signal_Next` signals the next node by setting its `spin` field.

In Figure 5, the superficial conflict happens between `Lock` and `Unlock` on the `spin` field. This conflict is more subtle than the one in the queue example, since `spin` is essential for correctness of the operations. Abstracting the actions accessing `spin` or hiding `spin` violates the mutual-exclusion property that the program must provide. The conflict prevents us from combining the pairs of actions in either procedure.

We resolve the superficial conflict by keeping enough information, in new variables, about when the `spin` is signaled. Figure 6 shows the annotated code with new variables. We introduce a new field `thread` to `Node`, which keeps track of the thread id that created the node. In addition `H` points to the head of the list, and `owner`

Implementation (*Impl*)

```
record Lelem { spin: int; next: Lelem; }
var L: Lelem; // tail of list

Unlock(I: Lelem)
Check_Empty: atomic {
  if (L == I) {
   L := null;
   return;
  }
 }
Signal_Next: atomic {
  I.next.spin := 0;
 }
```

```
Lock()
Create_Node: atomic {
  I := New Lelem;
  I.spin := 1;
  I.next := null;
  pred := L;
  L := I;
  if (pred != null)
   pred.next := I;
 }
Wait_Prev: atomic {
  if (pred != null)
   assume I.spin != 1;
 }
 return I;
```

Specification (*Spec*)

```
atomic Unlock(I: Lelem)
   assert I != null && owner == I;
   owner := null;
```

```
atomic Lock()
  I := New Lelem;
  assume owner == null;
  owner := I;
  return I;
```

Figure 5. The reduced implementation and the specification of the lock

```
Unlock(I: Lelem)
Check_Empty: atomic {
  if (L == I) {
*  assert I != null && owner == I;
   L := null;
*  owner := H := null;
   return;
  }
 }
Signal_Next: atomic {
* assert I != null && owner == I;
  I.next.spin := 0;
* H := I.next;
* owner := null;
 }
```

```
Lock()
Create_Node: atomic {
  I := New Lelem;
  I.spin := 1;
  I.next := null;
* I.thread := tid;
  pred := L;
  L := I;
  if (pred != null)
   pred.next := I;
  else
*  owner := H := I;
 }
Wait_Prev: atomic {
  if (pred != null) {
   assert I.thread == tid && Reach(next, H, I, L);
   assume I.spin != 1;
*  owner := I;
*  I.thread := 0;
  }
 }
 return I;
```

Figure 6. The lock implementation after introducing `.thread`, `H` and `owner`

points to the node holding the lock. Usually `H==owner`; however, when `Signal_Next` signals the next node, `H` is set to the signalled node, and `owner` is set to `null`. The following invariant allows to infer that when `Wait_Prev` in `Lock` sees `spin!=1`, `owner==null` and the current node `I` is the head, so it can acquire the node safely.

```
invariant (L == null || L.next == null)
 && (L == null <==> H == null)
 && Reach(next, H, L, null)
 && (owner == null || owner == H)
 && (forall u: Node :: Reach(next, H, u, L) && n != null
       ==> u.alloc && (u.spin == 1 || u == H))
 && (H != null && H.spin != 1 && H.thread != 0
       ==> owner == null)
```

At this point, we can hide the `spin` field, but now replace the instruction `assume (spin != 1)` in `Wait_Prev` with `assume (owner==null && H==I)`, which will make `Lock` wait until the node becomes the head of the list. Figure 5 shows the final version of the code after another reduction phase that combines the new actions, followed by a refinement phase that hides the unnecessary variables. Notice that, an assertion remains in `Unlock`, which states that the given node must be node that is holding the lock, created by a previous call to `Lock`. This assertion is in fact the pre-condition of `Unlock`; the linearizability proof will hold as long as the clients satisfy the precondition.

References

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

[2] Daphna Amit, Noam Rinetzky, Thomas W. Reps, Mooly Sagiv, and Eran Yahav. Comparison under abstraction for verifying linearizability. In *CAV*, pages 477–490, 2007.

[3] Robert Colvin, Lindsay Groves, Victor Luchangco, and Mark Moir. Formal verification of a lazy concurrent list-based set algorithm. In *CAV*, volume 4144 of *Lecture Notes in Computer Science*, pages 475–488. Springer, 2006.

[4] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. A calculus of atomic actions. Technical Report MSR-TR-2008-99, Microsoft Research Redmond, July 2008.

[5] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. A calculus of atomic actions. In *POPL '09: ACM Symposium on Principles of Programming Languages*, New York, NY, USA, 2009. ACM.

[6] Tayfun Elmas, Ali Sezgin, Serdar Tasiran, and Shaz Qadeer. An annotation assistant for interactive debugging of programs with common synchronization idioms. In *Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*, July 2009.

[7] H. Gao, J. F. Groote, and W. H. Hesselink. Lock-free dynamic hash tables with open addressing. *Distrib. Comput.*, 18(1):21–42, 2005.

[8] Lindsay Groves. Verifying michael and scott's lock-free queue algorithm using trace reduction. In *CATS '08: Symposium on Computing: the Australasian theory*, pages 133–142, Darlinghurst, Australia, 2008. Australian Computer Society, Inc.

[9] Danny Hendler, Nir Shavit, and Lena Yerushalmi. A scalable lock-free stack algorithm. In *SPAA '04: ACM symposium on Parallelism in algorithms and architectures*, pages 206–215, New York, NY, USA, 2004. ACM.

[10] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[11] Wim H. Hesselink. Eternity variables to prove simulation of specifications. *ACM Trans. Comput. Logic*, 6(1):175–201, 2005.

[12] Bengt Jonsson, Amir Pnueli, and Camilla Rump. Proving refinement using transduction. *Distrib. Comput.*, 12(2-3):129–149, 1999.

[13] Yonit Kesten, Amir Pnueli, Elad Shahar, and Lenore D. Zuck. Network invariants in action. In *CONCUR '02: The 13th International Conference on Concurrency Theory*, pages 101–115, London, UK, 2002. Springer-Verlag.

[14] Orran Krieger, Michael Stumm, Ron Unrau, and Jonathan Hanna. A fair fast scalable reader-writer lock. In *ICPP '93: The International Conference on Parallel Processing*, pages 201–204, Washington, DC, USA, 1993. IEEE Computer Society.

[15] Shuvendu Lahiri and Shaz Qadeer. Back to the future: revisiting precise program verification using smt solvers. In *POPL '08: ACM symposium on Principles of programming languages*, pages 171–182, New York, NY, USA, 2008. ACM.

[16] Richard J. Lipton. Reduction: a method of proving properties of parallel programs. *Commun. ACM*, 18(12):717–721, 1975.

[17] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC '96: ACM symposium on Principles of distributed computing*, pages 267–275, New York, NY, USA, 1996. ACM.

[18] Susan Owicki and David Gries. Verifying properties of parallel programs: an axiomatic approach. *Commun. ACM*, 19(5):279–285, 1976.

[19] Seungjoon Park and David L. Dill. Protocol verification by aggregation of distributed transactions. In *CAV '96: The International Conference on Computer Aided Verification*, pages 300–310, London, UK, 1996. Springer-Verlag.

[20] R.K.Treiber. Systems programming: Coping with parallelism. rj5118, April 1986.

[21] Viktor Vafeiadis. Shape-value abstraction for verifying linearizability. In *VMCAI '09: The International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 335–348, Heidelberg, 2009. Springer-Verlag.

[22] Viktor Vafeiadis, Maurice Herlihy, Tony Hoare, and Marc Shapiro. Proving correctness of highly-concurrent linearisable objects. In *PPoPP '06 ACM Symposium on Principles and practice of parallel programming*, pages 129–136, New York, NY, USA, 2006. ACM.

[23] Liqiang Wang and Scott D. Stoller. Static analysis for programs with non-blocking synchronization. In *ACM SIGPLAN 2005 Symposium on Principles and Practice of Parallel Programming (PPoPP)*. ACM Press, June 2005.

A. Proof of the soundness theorem

A.1 Notations, definitions

In the following we extend the definition of state transitions as follows:

- $\sigma_1 \xrightarrow{l} \sigma_2$ holds if there exists ϵ_1, ϵ_2 such that $(\sigma_1, \epsilon_1) \xrightarrow{l} (\sigma_2, \epsilon_2)$
- $\epsilon_1 \xrightarrow{l} \epsilon_2$ holds if there exists σ_1, σ_2 such that $(\sigma_1, \epsilon_1) \xrightarrow{l} (\sigma_2, \epsilon_2)$

Note that, $\sigma_1 \xrightarrow{l} \sigma_2$ and $\epsilon_1 \xrightarrow{l} \epsilon_2$ together imply $(\sigma_1, \epsilon_1) \xrightarrow{l} (\sigma_2, \epsilon_2)$.

DEFINITION 6 (Behaviorally-simulating runs). *Let P and P' be two programs with $\mathsf{Sig}(P) = \mathsf{Sig}(P')$, and let $\mathcal{I}$ be a state predicate. Let r and r' be maximal runs of P and P', respectively. The run r' is said to* behaviorally-simulate r*, denoted $r \blacktriangleleft r'$, if 1)* $\mathsf{Beh}(r) = \mathsf{Beh}(r')$*, and 2)* $\ll_r \subseteq \ll_{r'}$.

A.2 Proofs

For statement c, we define $c[\beta/\alpha]$ be the statement in which every occurrence of α is replaced by β. In addition, $\epsilon[\beta/\alpha]$ and $P[\beta/\alpha]$ are defined to be ϵ and P in which the same replacement is applied to every $\epsilon(t)$ and *body*(ρ), respectively.

The following lemma states that after replacing α with β in the program, the sequences of ϵ-transitions remain the same.

LEMMA 1. *Let P be a program. The following holds.*

- *For $t \neq u$, if $\epsilon_1 \xrightarrow{(t,\alpha)(u,\beta)} \epsilon_2$ be a run of P, then $\epsilon_1 \xrightarrow{(u,\beta)(t,\alpha)} \epsilon_2$ is a run of P.*
- *If $\epsilon_1 \xrightarrow{(t,\alpha)} \epsilon_2$ be a run of P, then $\epsilon_1[\beta/\alpha] \xrightarrow{(t,\alpha)} \epsilon_2[\beta/\alpha]$ is a run of $P[\beta/\alpha]$.*
- *If $\epsilon_1 \xrightarrow{(t,\alpha)(t,\beta)} \epsilon_2$ be a run of P, then $\epsilon_1[\gamma/(\alpha;\beta)] \xrightarrow{(t,\gamma)} \epsilon_2[\gamma/(\alpha;\beta)]$ is a run of $P[\gamma/(\alpha;\beta)]$.*
- *If $\epsilon_1 \xrightarrow{(t,\alpha)} \epsilon_2$ be a run of P, then $\epsilon_1[\gamma/(\alpha\Box\beta)] \xrightarrow{(t,\gamma)} \epsilon_2[\gamma/(\alpha\Box\beta)]$ is a run of $P[\gamma/(\alpha;\beta)]$.*
- *If $\epsilon_1 \xrightarrow{(t,\beta)} \epsilon_2$ be a run of P, then $\epsilon_1[\gamma/(\alpha\Box\beta)] \xrightarrow{(t,\gamma)} \epsilon_2[\gamma/(\alpha\Box\beta)]$ is a run of $P[\gamma/(\alpha;\beta)]$.*
- *If $\epsilon_1 \xrightarrow{(t,\alpha_1)\ldots(t,\alpha_n)} \epsilon_2$ be a run of P, then $\epsilon_1[\beta/\alpha^{\circlearrowleft}] \xrightarrow{(t,\beta)} \epsilon_2[\beta/\alpha^{\circlearrowleft}]$ is a run of $P[\beta/\alpha^{\circlearrowleft}]$.*

PROOF.

$\langle 1 \rangle 1.$ The claims of the lemma hold by the operational semantics of the language given in Section 3.
$\langle 1 \rangle 2.$ Q.E.D.

We will extensively use the following properties of $\blacktriangleleft$ in our proofs.

LEMMA 2. *$\blacktriangleleft$ is reflexive and transitive.*

PROOF.

$\langle 1 \rangle 1.$ Equivalence on Beh and $\subseteq$ relation on $\ll$ are both reflexive and transitive. Thus, $\blacktriangleleft$ is trivially reflexive and transitive.
$\langle 1 \rangle 2.$ Q.E.D.

The following lemma states that final state of a run determines the behavior of that run.

LEMMA 3 (Final state). *Let $(P, \mathcal{I})$ be a proof state such that P is good from $\mathcal{I}$. Let $r = r_1 \xrightarrow{\mathbf{l}} r_n$ and $r' = r'_1 \xrightarrow{\mathbf{l}'} r'_m$ are two maximal runs of P from $\mathcal{I}$ such that $Tid(r) = Tid(r')$ and every thread calls the same procedure in r and r'. If $\sigma_{r_n} = \sigma_{r'_m}$, then* $\mathsf{Beh}(r) = \mathsf{Beh}(r')$.

PROOF.

$\langle 1 \rangle 1.$ The behaviors are encoded by input and output variables of the threads, which are all local. For each thread $t \in Tid(r)$ let ρ be the procedure called by t. The input variables $\overrightarrow{in}_\rho$ are read-only, and output variables $\overrightarrow{out}_\rho$ are only written by t. Thus, the behavior $\mathsf{beh}(r, t)$ can be determined by only looking at the values $\overrightarrow{in}_\rho$ and $\overrightarrow{out}_\rho$ at r_n and r_m. Thus, the equivalence of the final states implies the equivalence of the behaviors.
$\langle 1 \rangle 2.$ Q.E.D.

LEMMA 4 (Right-mover). *Let $(P, \mathcal{I})$ be a proof state such that P is good from $\mathcal{I}$, and α be action that is right-mover. Let $r = r_1 \xrightarrow{\mathbf{l_1}} r_i \xrightarrow{(t,\alpha)\mathbf{l}} r_j \xrightarrow{(t,\gamma)\mathbf{l_2}} r_n$ be a maximal run of P from $\mathcal{I}$., such that $\mathbf{l} = (u_1, \beta_1)(u_2, \beta_2)\cdots(u_m, \beta_m)$ and $t \notin \{u_1, \ldots, u_m\}$. There exists a maximal run $r' = r_1 \xrightarrow{\mathbf{l_1}} r_i \xrightarrow{\mathbf{l}(t,\alpha)} r_j \xrightarrow{(t,\gamma)\mathbf{l_2}} r_n$ of P from $\mathcal{I}$, such that $r \blacktriangleleft r'$.*

PROOF.

ASSUME: $(P, \mathcal{I}) \vdash \alpha : \mathbb{R}$
$\langle 1 \rangle 1.$ Induction on the length (m) of the sequence $\mathbf{l} = (u_1, \beta_1)(u_2, \beta_2)\cdots(u_m, \beta_m)$
CASE: Base case

$\langle 2\rangle 1.$ The claim trivially holds when $m = 0$, by taking $r = r'$.
$\langle 2\rangle 2.$ Q.E.D.
CASE: Inductive case
$\langle 2\rangle 1.$ Consider the run $r = r_1 \xrightarrow{\mathbf{l_1}} r_i \xrightarrow{(t,\alpha)\mathbf{l}} r_k \xrightarrow{(u_{m+1},\beta_{m+1})} r_j \xrightarrow{(t,\gamma)\mathbf{l_2}} r_n$ such that $t \notin \{u_1, \ldots, u_m, u_{m+1}\}$.
$\langle 2\rangle 2.$ By the inductive hypothesis, there exists a run $r'' = r_1 \xrightarrow{\mathbf{l_1}} r_i \xrightarrow{\mathbf{l}(t,\alpha)} r_k \xrightarrow{(u_{m+1},\beta_{m+1})} r_j \xrightarrow{(t,\gamma)\mathbf{l_2}} r_n$ such that $r'' \blacktriangleleft r$.
PROVE: There exists $r' = r_1 \xrightarrow{\mathbf{l_1}} r_i \xrightarrow{\mathbf{l}(u_{m+1},\beta_{m+1})(t,\alpha)} r_j \xrightarrow{(t,\gamma)\mathbf{l_2}}$.
$\langle 3\rangle 1.$ Let r'' in $\langle 2\rangle 2$ be as follows: $r'' = r_1 \xrightarrow{\mathbf{l_1}} r_i \xrightarrow{\mathbf{l}} r_z \xrightarrow{(t,\alpha)} r_k \xrightarrow{(u_{m+1},\beta_{m+1})} r_j \xrightarrow{(t,\gamma)\mathbf{l_2}} r_n$
$\langle 3\rangle 2.$ $\sigma_{r_z} \vDash \phi_{\beta_{m+1}} \wedge \mathsf{wp}(\tau_{\beta_{m+1}}, \phi_\alpha)$, since P is good from $\mathcal{I}$.
$\langle 3\rangle 3.$ $\sigma_{r_z} \xrightarrow{(u_{m+1},\beta_{m+1})(t,\alpha)} \sigma_{r_j}$ holds, since $(P,\mathcal{I}) \vdash \alpha : \mathbb{R}$, and $\tau_{\beta_{m+1}} \cdot \tau_\alpha \Rightarrow \tau_\alpha \cdot \tau_{\beta_{m+1}}$.
$\langle 3\rangle 4.$ $\epsilon_{r_z} \xrightarrow{(u_{m+1},\beta m+1)(t,\alpha)} \epsilon_{r_j}$ holds, by $t \neq u_{m+1}$ and Lemma 1.
$\langle 3\rangle 5.$ $r_z \xrightarrow{(u_{m+1},\beta_{m+1})} r_s \xrightarrow{(t,\alpha)} r_j$ for some r_s by $\langle 3\rangle 3$ and $\langle 3\rangle 4$.
$\langle 3\rangle 6.$ Q.E.D.
PROVE: $r \blacktriangleleft r'$
$\langle 3\rangle 1.$ $r \blacktriangleleft r''$ by the inductive hypothesis ($\langle 2\rangle 2$).
PROVE: $r'' \blacktriangleleft r'$
$\langle 4\rangle 1.$ Since the final states of r'' and r' are equivalent, $\mathsf{Beh}(r'') = \mathsf{Beh}(r')$ by Lemma 3.
$\langle 4\rangle 2.$ The only way to break a relation in $\ll_{r''}$, by commuting (t,α) with (u_{m+1},β_{m+1}) is when α is the last action of t and β_{m+1} is the first action of u_{m+1}. However, this contradicts with the existence of γ. Thus $\ll_{r''} \subseteq \ll_{r'}$
$\langle 4\rangle 3.$ Q.E.D.
$\langle 3\rangle 2.$ $r \blacktriangleleft r'$ by $\langle 3\rangle 1$, $r'' \blacktriangleleft r'$, and $\blacktriangleleft$ is transitive.
$\langle 3\rangle 3.$ Q.E.D.
$\langle 1\rangle 2.$ Q.E.D.

LEMMA 5 (Left-mover). *Let $(P,\mathcal{I})$ be a proof state such that P is good from $\mathcal{I}$, and γ be action that is left-mover. Let $r = r_1 \xrightarrow{\mathbf{l_1}(t,\alpha)} r_i \xrightarrow{\mathbf{l}(t,\gamma)} r_j \xrightarrow{\mathbf{l_2}} r_n$ be a successful, maximal run of P from $\mathcal{I}$., such that $\mathbf{l} = (u_1,\beta_1)(u_2,\beta_2)\cdots(u_m,\beta_m)$ and $t \notin \{u_1, \ldots, u_n, u_{n+1}\}$. There exists a maximal run $r' = r_1 \xrightarrow{\mathbf{l_1}(t,\alpha)} r_i \xrightarrow{(t,\gamma)\mathbf{l}} r_j \xrightarrow{\mathbf{l_2}} r_n$ of P from $\mathcal{I}$, such that $r \blacktriangleleft r'$.*

PROOF.

$\langle 1\rangle 1.$ The proof is symmetric to that of Lemma 4.
$\langle 1\rangle 2.$ Q.E.D.

We now state and prove the main soundness theorem of this paper.

THEOREM 3 (Soundness). *Let $(P_1,\mathcal{I}_1) \dashrightarrow \cdots \dashrightarrow (P_n,\mathcal{I}_n)$ be a sequence of proof steps such that P_n is good from $\mathcal{I}_n$. Then forall $1 \le i \le n$, $P_1 \lhd_{\mathcal{I}_n} P_i$ holds.*

PROOF.

ASSUME: Since P_n is good from $\mathcal{I}_n$, forall $1 \le i \le n$, P_i is good from $\mathcal{I}_n$, by Theorem 1.
$\langle 1\rangle 1.$ Proof by induction on the length of the proof, i.e. the number of proof states reached during the proof.
CASE: Base case
$\langle 2\rangle 1.$ There is only one proof state $(P_1,\mathcal{I}_1)$.
$\langle 2\rangle 2.$ The claim holds trivially for $(P_1,\mathcal{I}_1)$, since $P_1 \lhd_{\mathcal{I}_1} P_1$, by definition of $\lhd$ and Lemma 2.
CASE: Inductive case
$\langle 2\rangle 1.$ Consider part of proof: $(P_1,\mathcal{I}_1) \dashrightarrow \cdots \dashrightarrow (P_i,\mathcal{I}_i) \dashrightarrow (P_{i+1},\mathcal{I}_{i+1}) \dashrightarrow \cdots \dashrightarrow (P_n,\mathcal{I}_n)$
ASSUME: Forall $1 \le j \le i$, $P_1 \lhd_{\mathcal{I}_n} P_j$ holds, by inductive hypothesis.
PROVE: $P_1 \lhd_{\mathcal{I}_n} P_{i+1}$.
$\langle 3\rangle 1.$ For any $1 \le k \le n$, let $X_k = \mathit{Var}_{P_k} \backslash \mathsf{fv}(\mathcal{I}_n)$.
ASSUME: A run $r = r_1 \xrightarrow{\mathbf{1}} r_N$ of P_1 from $\exists X_1.\mathcal{I}_n$.
PROVE: There exists a run r' of P_{n+1} from $\exists X_{i+1}.\mathcal{I}_n$ such that $r \blacktriangleleft r'$.
$\langle 4\rangle 1.$ Case split by the proof rule.
CASE: Rule INVARIANT: Replace invariant $\mathcal{I}_1$ with $\mathcal{I}_2$ if $\alpha \leftrightarrows \mathcal{I}_2$ for all the actions α in P, and $\mathcal{I}_2 \Rightarrow \mathcal{I}_1$.
$\langle 5\rangle 1.$ $P_i = P_{i+1}$, so taking $r' = r$ makes the claim hold, since $r \blacktriangleleft r$ by Lemma 2.
$\langle 5\rangle 2.$ Q.E.D.
CASE: Rule ABSTRACT: Replace the action α with action β if $\beta \leftrightarrows \mathcal{I}$ and $(P,\mathcal{I}) \vdash \alpha \preceq \beta$.
ASSUME: $(P_i,\mathcal{I}_i) \vdash \alpha \preceq \beta$.
$\langle 5\rangle 1.$ $\mathcal{I}_{n+1} = \mathcal{I}_n$, by proof rule.
$\langle 5\rangle 2.$ $X_i = X_{i+1}$, since $\mathit{Var}_{P_i} = \mathit{Var}_{P_{i+1}}$, by proof rule.
$\langle 5\rangle 3.$ Construct r' from r inductively in the following way.
CASE: Base case

⟨6⟩1. Take $r'_1 = (\sigma_{r_1}, \epsilon_{r_1}[\beta/\alpha])$.

CASE: Inductive case

ASSUME: $r'_1 \longrightarrow r'_j$ so that $\forall 1 \leq k \leq j.\ r'_j = (\sigma_{r_j}, \epsilon_{r_j}[\beta/\alpha])$.

ASSUME: Transition $r_j \xrightarrow{(t,\gamma)} r_{j+1}$.

PROVE: $(\sigma_{r_j}, \epsilon_{r_j}[\beta/\alpha]) \xrightarrow{(t,\gamma)} (\sigma_{r_{j+1}}, \epsilon_{r_{j+1}}[\beta/\alpha])$

⟨7⟩1. If $\gamma \neq \alpha$, the claim trivially holds by operational semantics of atomic actions, and Lemma 1.

⟨7⟩2. Let $\gamma = \alpha$.

⟨7⟩3. $\sigma_{r_j} \xrightarrow{(t,\beta)} \sigma_{r_{j+1}}$, since $(P, \mathcal{I}) \vdash \alpha \preceq \beta$, and P is good from $\mathcal{I}$.

⟨7⟩4. $\epsilon_{r_j}[\beta/\alpha] \xrightarrow{(t,\beta)} \epsilon_{r_{j+1}}[\beta/\alpha]$, by Lemma 1.

⟨7⟩5. $(\sigma_{r_j}, \epsilon_{r_j}[\beta/\alpha]) \xrightarrow{(t,\alpha)} (\sigma_{r_{j+1}}, \epsilon_{r_{j+1}}[\beta/\alpha])$ by ⟨7⟩3 and ⟨7⟩4.

⟨7⟩6. Q.E.D.

PROVE: $r \blacktriangleleft r'$

⟨6⟩1. By the above construction, $Tid(r) = Tid(r')$, and every procedure executes the same procedure in both r and r'.

⟨6⟩2. By the above construction, $\forall 1 \leq j \leq N.\ r_j = r'_j$.

⟨6⟩3. $\mathsf{Beh}(r) = \mathsf{Beh}(r')$ by ⟨6⟩1, ⟨6⟩2, and Lemma 3.

⟨6⟩4. For every thread $t \in Tid(r)$, $\mathsf{fst}(r,t) = \mathsf{fst}(r',t)$ and $\mathsf{lst}(r,t) = \mathsf{lst}(r',t)$. Thus $\ll_r = \ll_{r'}$

⟨6⟩5. Q.E.D.

⟨5⟩4. By ⟨5⟩2, $r_1 \vDash \exists X_n.\mathcal{I}_n$ implies $r'_1 \vDash \exists X_{i+1}.\mathcal{I}_n$. Thus r' is a run of P_{i+1} from $\exists X_{i+1}.\mathcal{I}_n$ such that $r \blacktriangleleft r'$.

⟨5⟩5. Q.E.D.

CASE: Rule REDUCE-SEQUENTIAL: Replace occurrences of $\alpha\ ;\ \gamma$ with $[\![\alpha\ ;\ \gamma]\!]$ if either $(P, \mathcal{I}) \vdash \alpha : \mathbb{R}$ or $(P, \mathcal{I}) \vdash \gamma : \mathbb{L}$.

⟨5⟩1. $\mathcal{I}_{n+1} = \mathcal{I}_n$ by proof rule

⟨5⟩2. $X_i = X_{i+1}$, since $Var_{P_i} = Var_{P_{i+1}}$ by prood rule.

⟨5⟩3. We do the proof for the cases in which α is right-mover and γ is left-mover separately.

CASE: $(P_i, \mathcal{I}_i) \vdash \alpha : \mathbb{R}$

⟨6⟩1. Construct r'', a run of P_i from r, such that $r \blacktriangleleft r''$ inductively (on the number of (α, γ) pairs not appearing adjacent in r).

CASE: Base case

⟨8⟩1. Take $r'' = r$, so $r \blacktriangleleft r''$.

CASE: Inductive case

ASSUME: Run $r'' = r_1 \xrightarrow{\mathbf{l_1}} r_i \xrightarrow{(t,\alpha)\mathbf{l}(t,\gamma)} r_j \xrightarrow{\mathbf{l_2}} r_N$, such that $\mathbf{l} = (u_1, \beta_1)(u_2, \beta_2) \cdots (u_m, \beta_m)$, $t \notin \{u_1, \ldots, u_m, u_{m+1}\}$, and $r \blacktriangleleft r''$

⟨8⟩1. $r''' = r_1 \xrightarrow{\mathbf{l_1}} r_i \xrightarrow{\mathbf{l}(t,\alpha)(t,\gamma)} r_j \xrightarrow{\mathbf{l_1}} r_n$ exists such that $r'' \blacktriangleleft r''$, by Lemma 4.

⟨8⟩2. The number of pairs α, γ in r''' is one less than r''.

⟨7⟩1. Output run r'' such that for every $r''(k) = (t, \alpha)$ for some thread t, $r''(k+1) = (t, \gamma)$, i.e., all (α, γ) pairs appear adjacent to each other. As a result of the above induction, $r \blacktriangleleft r''$.

⟨7⟩2. Q.E.D.

⟨6⟩2. Construct r', a run of P_{i+1} from r'' inductively (on the length of the run).

CASE: Base case

⟨8⟩1. Take $r'_1 = r''_1$.

CASE: Inductive case

ASSUME: $r' = r''_1 \xrightarrow{\mathbf{l}} r''_k$ for some $\mathbf{l}$.

ASSUME: Transition $r''_k \xrightarrow{(t,\alpha)} r''_{k+1} \xrightarrow{(t,\gamma)} r''_{k+2}$.

PROVE: $(\sigma_{r''_k}, \epsilon_{r''_k}[[\![\alpha;\gamma]\!]/\alpha;\beta]) \xrightarrow{(t,[\![\alpha;\gamma]\!])} (\sigma_{r''_{k+2}}, \epsilon_{r''_{k+2}}[[\![\alpha;\gamma]\!]/\alpha;\gamma])$.

⟨9⟩1. $[\![\alpha;\gamma]\!] = \mathsf{assert}\,(\phi_\alpha \wedge \mathsf{wp}(\tau_\alpha, \phi_\gamma)); (\tau_\alpha \cdot \tau_\gamma)$, by definition of $[\![\ ]\!]$.

⟨9⟩2. Since the run r is successful, $r''_k \vDash \phi_\alpha$, and in addition $r''_{k+1} \vDash \phi_\gamma$.

⟨9⟩3. $r''_k \vDash \mathsf{wp}(\tau_\alpha, \phi_\gamma)$

⟨9⟩4. $(\tau_\alpha(r_k, r''_{k+1}) \wedge \tau_\gamma(r''_{k+1}, r''_{k+2})) \Rightarrow (\tau_\alpha \cdot \tau_\gamma)(r''_k, r''_{k+2})$, by definition of $\cdot$.

⟨9⟩5. $\sigma_{r''_k} \xrightarrow{(t,[\![\alpha;\gamma]\!])} \sigma_{r''_{k+2}}$

⟨9⟩6. $\epsilon_{r''_k}[[\![\alpha;\gamma]\!]/\alpha;\gamma] \xrightarrow{(t,[\![\alpha;\gamma]\!])} \epsilon_{r''_{k+2}}[[\![\alpha;\gamma]\!]/\alpha;\gamma]$, by Lemma 1.

⟨9⟩7. Q.E.D.

PROVE: $r'' \blacktriangleleft r'$

⟨8⟩1. By the above construction, $Tid(r'') = Tid(r')$, and every procedure executes the same procedure in both r'' and r'.

⟨8⟩2. By the above construction, $r''_N = r'_N$.

⟨8⟩3. $\mathsf{Beh}(r'') = \mathsf{Beh}(r')$ by ⟨8⟩1, ⟨8⟩2, and Lemma 3.

⟨8⟩4. $\ll_{r''} = \ll_{r'}$, since the above construction does not introduce in r' any more interleavings than that of r'' between two threads.

⟨8⟩5. Q.E.D.

⟨6⟩3. $r'_1 \vDash \exists X_{i+1}.\mathcal{I}_n$, by $\sigma_{r_1} = \sigma_{r''_1} = \sigma_{r'_1}$, $r_1 \vDash \exists X_i.\mathcal{I}_n$, and ⟨5⟩2. Thus r' is a run of P_{i+1} from $\exists X_{i+1}.\mathcal{I}_n$.

$\langle 6\rangle 4.$ $r \blacktriangleleft r'$, by $r \blacktriangleleft r''$, $r'' \blacktriangleleft r'$, and Lemma 2.
$\langle 6\rangle 5.$ Q.E.D.
CASE: $(P_i, \mathcal{I}_i) \vdash \gamma : \mathbb{L}$
$\langle 6\rangle 1.$ This case is proved similar to the former case, using Lemma 5.
$\langle 6\rangle 2.$ Q.E.D.
$\langle 5\rangle 4.$ Q.E.D.
CASE: Rule REDUCE-CHOICE: Replace occurrences of $\alpha \,\square\, \gamma$ with $[\![\alpha \,\square\, \gamma]\!]$.
$\langle 5\rangle 1.$ $\mathcal{I}_{n+1} = \mathcal{I}_n$, by proof rule
$\langle 5\rangle 2.$ $X_i = X_{i+1}$, since $Var_{P_i} = Var_{P_{i+1}}$, by proof rule.
$\langle 5\rangle 3.$ Construct r', a run of P_{i+1} from r inductively (on the length of the run).
CASE: Base case
$\langle 7\rangle 1.$ Take $r'_1 = r_1$.
CASE: Inductive case
ASSUME: $r' = r_1 \xrightarrow{\mathbf{l}} r_k$ for some $\mathbf{l}$.
ASSUME: $r_k \xrightarrow{(t,\beta)} r_{k+1}$
PROVE: $(\sigma_{r_k}, \epsilon_{r_k}[[\![\alpha\square\gamma]\!]/\alpha;\gamma]) \xrightarrow{(t,\beta)} (\sigma_{r_{k+1}}, \epsilon_{r_{k+1}}[[\![\alpha\square\gamma]\!]/\alpha\square\gamma])$
$\langle 8\rangle 1.$ If $\beta \neq \alpha$ and $\beta \neq \gamma$, then the claim trivially holds by operational semantics of atomic actions, and Lemma 1.
$\langle 8\rangle 2.$ Let $\beta = \alpha$ (where the transition is due to $\alpha\square\gamma$).
$\langle 8\rangle 3.$ $[\![\alpha\square\gamma]\!] = \mathsf{assert}\,(\phi_\alpha \wedge \phi_\gamma); (\tau_\alpha \vee \tau_\gamma)$, by definition of $[\![\;]\!]$.
$\langle 8\rangle 4.$ Since P_i is good from $\exists X_i.\mathcal{I}$, $r_k \vDash \phi_\gamma$, and in addition to $r_k \vDash \phi_\alpha$.
$\langle 8\rangle 5.$ $\tau_\alpha(r_k, r_{k+1}) \Rightarrow (\tau_\alpha \vee \tau_\gamma)(r_k, r_{k+1})$.
$\langle 8\rangle 6.$ $\sigma_{r_k} \xrightarrow{(t,[\![\alpha\square\gamma]\!])} \sigma_{r_{k+1}}$
$\langle 8\rangle 7.$ $\epsilon_{r_k}[[\![\alpha\square\gamma]\!]/\alpha\square\gamma] \xrightarrow{(t,[\![\alpha\square\gamma]\!])} \epsilon_{r_{k+1}}[[\![\alpha\square\gamma]\!]/\alpha\square\gamma]$, by Lemma 1.
$\langle 8\rangle 8.$ Similar construction is done if $\beta = \gamma$ (where the transition is due to $\alpha\square\gamma$).
$\langle 8\rangle 9.$ Q.E.D.
PROVE: $r \blacktriangleleft r'$
$\langle 7\rangle 1.$ By the above construction, $Tid(r) = Tid(r')$, and every procedure executes the same procedure in both r and r'.
$\langle 7\rangle 2.$ By the above construction, $r_N = r'_N$.
$\langle 7\rangle 3.$ $\mathsf{Beh}(r) = \mathsf{Beh}(r')$ by $\langle 7\rangle 1$, $\langle 7\rangle 2$, and Lemma 3.
$\langle 7\rangle 4.$ $\ll_r = \ll_{r'}$, since the above construction does not introduce in r' any more interleavings than that of r between two threads.
$\langle 7\rangle 5.$ Q.E.D.
$\langle 5\rangle 4.$ $r'_1 \vDash \exists X_{i+1}.\mathcal{I}_n$, by $\sigma_{r_1} = \sigma_{r''_1} = \sigma_{r'_1}$, $r_1 \vDash \exists X_i.\mathcal{I}_n$, and $\langle 5\rangle 2$. Thus r' is a run of P_{i+1} from $\exists X_{i+1}.\mathcal{I}_n$.
$\langle 5\rangle 5.$ Q.E.D.
CASE: Rule REDUCE-LOOP
$\langle 5\rangle 1.$ $\mathcal{I}_{n+1} = \mathcal{I}_n$
$\langle 5\rangle 2.$ $X_i = X_{i+1}$, since $Var_{P_i} = Var_{P_{i+1}}$.
CASE: $(P_i, \mathcal{I}_i) \vdash \alpha : \mathbb{R}$
$\langle 6\rangle 1.$ Construct r'', a run of P_i from r in the following way: For each occurrence of the pair α, γ, derive a new run from the current one, by moving every α (due to $\alpha^{\circlearrowleft}$) to the right, until it becomes adjacent to the last occurrence of α.
PROVE: $r \blacktriangleleft r''$
$\langle 7\rangle 1.$ By induction on the number of pairs α^1, α^2, where α^2 is last iteration of the loop, that do not appear adjacent in r.
$\langle 7\rangle 2.$ In the base case we take $r'' = r$, so $r \blacktriangleleft r''$.
$\langle 7\rangle 3.$ In the inductive case, we take the run $r = r_1 \xrightarrow{\mathbf{l_1}} r_i \xrightarrow{(t,\alpha^1)\mathbf{l}(t,\alpha^2)} r_j \xrightarrow{\mathbf{l_2}} r_n$ such that $\mathbf{l} = (u_1, \beta_1)(u_2, \beta_2)\cdots(u_m, \beta_m)$ and $t \notin \{u_1, \ldots, u_n, u_{n+1}\}$, and construct the run $r'' = r_1 \xrightarrow{\mathbf{l_1}} r_i \xrightarrow{\mathbf{l}(t,\alpha^1)(t,\alpha^2)} r_j \xrightarrow{\mathbf{l_1}} r_n$. By Lemma 4, $r \blacktriangleleft r''$ holds. In addition, the number of pairs α, γ in r'' is less than r.
$\langle 7\rangle 4.$ Output r'' such that for every $r''(k) = (t, \alpha)$ for some thread t, either $r''(k)$ is the last iteration, or $r''(k+1) = (t, \alpha)$. As a result of the above induction, $r \blacktriangleleft r''$.
$\langle 7\rangle 5.$ Q.E.D.
$\langle 6\rangle 2.$ Construct r', a run of P_{i+1} from r'' in the following way: Replace every occurrence of $r''_k \xrightarrow{(t,\alpha)^*} r''_{k+m}$ with $(\sigma_{r''_k}, \epsilon_{r''_k}[\alpha^{\circlearrowleft}/\beta]) \xrightarrow{(t,\beta)} (\sigma_{r''_{k+m}}, \epsilon_{r''_{k+m}}[\alpha^{\circlearrowleft}/\beta])$.
$\langle 7\rangle 1.$ Since τ_β is reflexive, in the case of $m = 0$, we yield the transition $(\sigma_{r''_k}, \epsilon_{r''_k}[\alpha^{\circlearrowleft}/\beta]) \xrightarrow{(t,\beta)} (\sigma_{r''_k}, \epsilon_{r''_k}[\alpha^{\circlearrowleft}/\beta])$
$\langle 7\rangle 2.$ In the case of $m > 0$, by $(P_i, \mathcal{I}_i) \vdash [\![\beta;\alpha]\!] \preceq \beta$, and since P_i is good from $\exists X_i.\mathcal{I}_n$,$\sigma_{r''_k} \xrightarrow{(t,[\![\alpha;\gamma]\!])} \sigma_{r''_{k+m}}$ holds.
$\langle 7\rangle 3.$ $\epsilon_{r''_k}[\alpha^{\circlearrowleft}/\beta] \xrightarrow{(t,\beta)} \epsilon_{r''_{k+m}}[\alpha^{\circlearrowleft}/\beta]$, by operational semantics.
$\langle 7\rangle 4.$ Q.E.D.
PROVE: $r'' \blacktriangleleft r'$
$\langle 7\rangle 1.$ $Tid(r'') = Tid(r')$.
$\langle 7\rangle 2.$ Since the final states of r'' and r' are equivalent, $\mathsf{Beh}(r'') = \mathsf{Beh}(r')$ by Lemma 3.

⟨7⟩3. For every thread $t \in Tid(r'')$, replacing consecutive transitions of t with again transitions of t does not effect relative ordering of t with other threads. Thus $\ll_r = \ll_{r'}$

⟨7⟩4. Q.E.D.

⟨6⟩3. $r_1 = r''_1$, by construction and Lemma 4.

⟨6⟩4. $r'_1 \models \exists X_{i+1}.\mathcal{I}_n$, by $\sigma_{r''_1} = \sigma_{r'_1}$, $r_1 \models \exists X_i.\mathcal{I}_n$, and ⟨5⟩2. Thus r' is a run of P_{i+1} from $\exists X_{i+1}.\mathcal{I}_n$.

⟨6⟩5. $r \blacktriangleleft r'$, by $r \blacktriangleleft r''$ and $r'' \blacktriangleleft r'$.

⟨6⟩6. Q.E.D.

CASE: $(P_i, \mathcal{I}_i) \vdash \alpha : \mathbb{L}$

⟨6⟩1. This case is proved similar to the former case, using Lemma 5.

⟨6⟩2. Q.E.D.

⟨5⟩3. Q.E.D.

CASE: Rule ADD-VARIABLE: Add the new variable v to Var_P, and replace every action α with β whenever $(P, \mathcal{I}) \vdash \alpha \preceq_{+v} \beta$, which holds if the following are both valid:

$$\mathbf{A1}.\quad (\mathcal{I} \wedge \neg\phi_\alpha) \quad\Rightarrow\quad (\forall v.\, \neg\phi_\beta) \qquad\qquad \mathbf{A2}.\quad (\mathcal{I} \wedge \tau_\alpha) \quad\Rightarrow\quad (\forall v.\, \neg\phi_\beta \vee (\exists v'.\, \tau_\beta))$$

⟨5⟩1. $\mathcal{I}_{n+1} = \mathcal{I}_n$, by proof rule.

⟨5⟩2. If v is a local variable, then $X_i = X_{i+1}$; otherwise $X_i = X_{i+1} \cup \{v\}$, since $Var_{P_{i+1}} = Var_{P_i} \cup \{v\}$.

⟨5⟩3. Construct r' from r inductively (inductive on the number of states in r) way.

CASE: Base case

⟨7⟩1. If $v \in \mathit{local}_\rho$ for some procedure ρ, then take $r'_1 = (\overline{\sigma}_{r_1}, \epsilon_{r_1}[\beta/\alpha])$, where $\overline{\sigma}_{r_1} = \sigma_{r_1}[t, v \mapsto x]$ and x is some value, for every thread t calling ρ.

⟨7⟩2. If v is global, then take $r'_1 = (\overline{\sigma}_{r_1}, \epsilon_{r_1}[\beta/\alpha])$, where $\overline{\sigma}_{r_1} = \sigma_{r_1}[v \mapsto x]$ such that $\overline{\sigma}_{r_1} \models \exists X_{i+1}.\mathcal{I}_n$. There exists x since $\sigma_{r_1} \models X_i.\mathcal{I}_n$ and $X_i = X_{i+1} \cup \{v\}$.

CASE: Inductive case

ASSUME:
$r' = r'_1 \xrightarrow{1} r'_k$ so that forall $1 \le j \le k$, $r'_j = (\overline{\sigma}_{r_j}, \epsilon_{r_j}[\beta/\alpha])$, where $\overline{\sigma}_{r_j} = \sigma_{r_j}[v \mapsto x]$, with x chosen during the construction properly.

ASSUME: Transition $r_k \xrightarrow{(t,\alpha)} r_{k+1}$ exists.

PROVE: $r'_k \xrightarrow{(t,\alpha)} (\sigma_{r_{k+1}}, \epsilon_{r_{k+1}}[\beta/\alpha])$

⟨8⟩1. $\overline{\sigma}_{r_k} \xrightarrow{(t,\beta)} \overline{\sigma}_{r_{k+1}}$, where $\overline{\sigma}_{r_{k+1}} = \sigma_{r_{k+1}}[v \mapsto x]$ such that $\tau_\beta(\overline{\sigma}_{r_k}, \overline{\sigma}_{r_{k+1}})$ holds. There exists x since $\tau_\alpha(\sigma_{r_k}, \sigma_{r_{k+1}}$, and by condition $\mathbf{A2}$ of $\preceq_{+v}$.

⟨8⟩2. $\epsilon_{r_k}[\beta/\alpha] \xrightarrow{(t,\beta)} \epsilon_{r_{k+1}}[\beta/\alpha]$, by Lemma 1.

PROVE: $r \blacktriangleleft r'$

⟨6⟩1. $Tid(r) = Tid(r')$.

⟨6⟩2. Let $V = Var_{P_i} \cap Var_{P_{i+1}}$. The above construction ensures that $\sigma_{r_N}|_V = \sigma_{r'_N}|_V$, and $\forall \rho.\ v \notin \mathit{local}_\rho$, thus $\mathsf{Beh}(r) = \mathsf{Beh}(r')$ by Lemma 3.

⟨6⟩3. For every thread $t \in Tid(r)$, $\mathsf{fst}(r, t) = \mathsf{fst}(r', t)$ and $\mathsf{lst}(r, t) = \mathsf{lst}(r', t)$. Thus $\ll_r = \ll_{r'}$.

⟨6⟩4. Q.E.D.

⟨5⟩4. If v is local, then $r'_1 \models \exists X_{i+1}.\mathcal{I}_n$ since $r_1 \models \exists X_i.\mathcal{I}_n$ and $X_i = X_{i+1}$.

⟨5⟩5. If v is global, $r'_1 \models \exists X_{i+1}.\mathcal{I}_n$ is ensured by the above construction, choosing $\sigma_{r'_1}(v)$ properly to satisfy $\exists X_i.\mathcal{I}_n$.

⟨5⟩6. Thus r' is a run of P_{i+1} from $\exists X_{i+1}.\mathcal{I}_n$.

⟨5⟩7. Q.E.D.

CASE: Rule HIDE-VARIABLE: Remove the existing variable v from the program, and replace the invariant $\mathcal{I}$ with $\exists v.\, \mathcal{I}$. Replace every action α with β whenever $(P, \mathcal{I}) \vdash \alpha \preceq_{-v} \beta$, which holds if the following are both valid:

$$\mathbf{H1}.\quad (\exists v.\, \mathcal{I} \wedge \neg\phi_\alpha) \quad\Rightarrow\quad \neg\phi_\beta \qquad\qquad \mathbf{H2}.\quad (\exists v, v'.\, \mathcal{I} \wedge \tau_\alpha) \quad\Rightarrow\quad (\neg\phi_\beta \vee \tau_\beta)$$

⟨5⟩1. $\mathcal{I}_{n+1} = \exists v.\mathcal{I}_n$, by proof rule.

⟨5⟩2. $X_i = X_{i+1}$, by proof rule.

⟨5⟩3. Construct r' from r inductively (inductive on the number of states in r) way.

CASE: Base case

⟨7⟩1. Take $r'_1 = (\overline{\sigma}_{r_1}, \epsilon_{r_1}[\beta/\alpha])$, where $\overline{\sigma}_{r_1} = \sigma_{r_1}|_{Var_{P_i}}$. $r_1 \models X_i.\mathcal{I}_n$, ⟨5⟩1 and ⟨5⟩2 imply that $r'_1 \models X_{i+1}.\mathcal{I}_n$.

CASE: Inductive case

ASSUME:
$r' = r'_1 \xrightarrow{1} r'_k$ so that forall $1 \le j \le k$, $r'_j = (\overline{\sigma}_{r_j}, \epsilon_{r_j}[\beta/\alpha])$, where $\overline{\sigma}_{r_j} = \sigma_{r_j}|_{Var_{P_i}}$.

ASSUME: Transition $r_k \xrightarrow{(t,\alpha)} r_{k+1}$ exists.

PROVE: $r'_k \xrightarrow{(t,\alpha)} (\sigma_{r_{k+1}}, \epsilon_{r_{k+1}}[\beta/\alpha])$

⟨8⟩1. $\overline{\sigma}_{r_k} \xrightarrow{(t,\beta)} \overline{\sigma}_{r_{k+1}}$, where $\overline{\sigma}_{r_{k+1}} = \sigma_{r_{k+1}}|_{Var_{P_i}}$, since condition $\mathbf{H2}$ of $\preceq_{-v}$ implies that if $\sigma_{r_k} \xrightarrow{(t,\alpha)} \sigma_{r_{k+1}}$ holds then $\sigma_{r_k}|_{Var_{P_i}} \xrightarrow{(t,\beta)} \sigma_{r_{k+1}}|_{Var_{P_i}}$ holds.

⟨8⟩2. $\epsilon_{r_k}[\beta/\alpha] \xrightarrow{(t,\beta)} \epsilon_{r_{k+1}}[\beta/\alpha]$, by Lemma 1.

PROVE: $r \blacktriangleleft r'$

⟨6⟩1. $Tid(r) = Tid(r')$.

⟨6⟩2. Let $V = Var_{P_i} \cap Var_{P_{i+1}}$. The above construction ensures that $\sigma_{r_N}|_V = \sigma_{r'_N}|_V$, and $\forall \rho.\ v \notin$ *local*$_\rho$, thus $\mathsf{Beh}(r) = \mathsf{Beh}(r')$ by Lemma 3.
⟨6⟩3. For every thread $t \in Tid(r)$, $\mathsf{fst}(r, t) = \mathsf{fst}(r', t)$ and $\mathsf{lst}(r, t) = \mathsf{lst}(r', t)$. Thus $\ll_r = \ll_{r'}$.
⟨6⟩4. Q.E.D.
⟨5⟩4. Since $r_1 \vDash \exists X_i.\mathcal{I}_n$ and $X_i = X_{i+1}$, the above construction ensures that $r'_1 \vDash \exists X_{i+1}.\mathcal{I}_n$ holds. Thus r' is a run of P_{i+1} from $\exists X_{i+1}.\mathcal{I}_n$.
⟨5⟩5. Q.E.D.
⟨1⟩2. Q.E.D.

B. Soundness of variable introduction and hiding

In the following we will give the proof of the preservation lemma for the ADD-VARIABLE and HIDE-VARIABLE rules. The proof of other rules and the proof of Theorem 1 can be easily adapted from the soundness proof in [4], which is given at the end of this paper (Section C).

LEMMA 6. *Let $(P_1, \mathcal{I}_1) \dashrightarrow (P_2, \mathcal{I}_2)$ be a proof steps. Let $V = Var_{P_1} \cap Var_{P_2}$ and $X = (Var_{P_1} \cup Var_{P_2}) \setminus V$. The following hold:*

C1. $Bad|_V(P_1, \exists X. \mathcal{I}_2) \subseteq Bad|_V(P_2, \exists X. \mathcal{I}_2)$ **C2**. $\forall (s_1, s_n) \in Good|_V(P_1, \exists X. \mathcal{I}_2)$:
a. $s_1 \in Bad|_V(P_2, \exists X. \mathcal{I}_2)$ *or* **b**. $(s_1, s_n) \in Good|_V(P_2, \exists X. \mathcal{I}_2)$

PROOF.

$\langle 1 \rangle 1$. Case split on the proof rule.

CASE: Rule ADD-VARIABLE: Add the new variable v to Var_P, and replace every action α with β whenever $(P, \mathcal{I}) \vdash \alpha \preceq_{+v} \beta$, which holds if the following are both valid:

$$\mathbf{A1}.\quad (\mathcal{I} \land \neg\phi_\alpha) \;\Rightarrow\; (\forall v.\, \neg\phi_\beta) \qquad \mathbf{A2}.\quad (\mathcal{I} \land \tau_\alpha) \;\Rightarrow\; (\forall v.\, \neg\phi_\beta \lor (\exists v'.\, \tau_\beta))$$

ASSUME: $\mathcal{I}_i = \mathcal{I}_{i+1}$, by proof rule. Call this invariant $\mathcal{I}$ in the following.
ASSUME: $X = \{v\}$, by proof rule. Use v instead of X in the formulas.
ASSUME: $V = Var_{prog_i}$, since $Var_{P_{i+1}} = Var_{P_i} \cup \{v\}$.
$\langle 2 \rangle 1$. $\mathcal{I}$ does not refer to v by proof rule. Thus $\mathcal{I} \iff \exists v.\mathcal{I}$.
PROVE: **C1**.
$\langle 3 \rangle 1$. Take a run $r = r_1 \xrightarrow{1} r_N = \mathsf{error}$ of P_1 such that $r_1 \vDash \exists X.\mathcal{I}$.
$\langle 3 \rangle 2$. Construct the run r' inductively.
CASE: Base case
$\langle 5 \rangle 1$. Take $r'_1 = (\overline{\sigma}_{r_1}), \epsilon_{r_1}[\beta/\alpha]$ such that $\overline{\sigma}_{r_1} = \sigma_{r_1}[v \mapsto x]$.
$\langle 5 \rangle 2$. By construction and $r_1 \vDash \exists X.\mathcal{I}$, $r'_1 \vDash \exists X.\mathcal{I}$ holds.
CASE: Inductive case
ASSUME: $r'_1 = r'_1 \xrightarrow{1'} r'_k$ such that $\forall 1 \leq j \leq k. r'_j = (\overline{\sigma}_{r_j}, \epsilon_{r_j}[\beta/\alpha])$, $\overline{\sigma}_{r_j} = \sigma_{r_j}[v \mapsto x]$ for some x, i.e., $\overline{\sigma}_{r_j}|_V = \sigma_{r_j}$.
$\langle 5 \rangle 1$. Take the transition $r_k \xrightarrow{(t,\alpha)} r_{k+1}$. There are two cases $r_{k+1} = \mathsf{error}$ or not.
ASSUME: $r_k \xrightarrow{(t,\alpha)} \mathsf{error}$.
$\langle 5 \rangle 2$. By condition $\mathbf{A1}$, if $\sigma_{r_k} \vDash \neg\phi_\alpha[t]$, then $\overline{\sigma}_{r_k} \vDash \neg\phi_\beta[t]$ for any value of $\overline{\sigma}_{r_k}(v)$. Thus $r'_k \xrightarrow{(t,\alpha)} \mathsf{error}$.
ASSUME: $r_k \xrightarrow{(t,\alpha)} r_{k+1}$ and $r_{k+1} \neq \mathsf{error}$.
$\langle 5 \rangle 3$. By condition $\mathbf{A2}$, if $\sigma_{r_k} \vDash \phi_\alpha[t]$ and $\tau_\alpha(\sigma_{r_k}, \sigma_{k+1})$ is true, then for any value of $\overline{\sigma}_{r_k}(v)$, either $\overline{\sigma}_{r_k} \vDash \neg\phi_\beta[t]$ or $\tau_\beta(\overline{\sigma}_{r_k}, \overline{\sigma}_{r_{k+1}})$ is true for some value $\overline{\sigma}_{r_{k+1}}(v)$. In the former case, we reach a violation, satisfying $\mathbf{C1}$, or obtain the run $r' = r'_1 \longrightarrow \mathbf{1}'(t, \beta) r'_{k+1}$ where $r'_{k+1} = (\overline{\sigma}_{r_{k+1}}, \epsilon_{r_{k+1}}[\beta/\alpha])$ and $\overline{\sigma}_{r_{k+1}}|_V = \sigma_{r_{k+1}}$.
$\langle 3 \rangle 3$. By the above construction, we obtain a run $r' = r'_1 \xrightarrow{1} r'_M = \mathsf{error}$ such that $M \leq N$. The case $r_N = \mathsf{error}$ ensures that the above construction reaches a violation when translating the transition $r_{N-1} \xrightarrow{(t,\alpha)} r_N = \mathsf{error}$ at the latest.
$\langle 3 \rangle 4$. Then $(\overline{\sigma}_{r_1}|_V) \in Bad(P_{i+1}, \exists X.\mathcal{I})$.
$\langle 3 \rangle 5$. Q.E.D.
PROVE: **C2**.
$\langle 3 \rangle 1$. Take a run $r = r_1 \xrightarrow{1} r_N$ where $r_N \neq \mathsf{error}$ of P_1 such that $r_1 \vDash \exists X.\mathcal{I}$.
$\langle 3 \rangle 2$. We construct the run r' inductively, similarly to the above case ($\mathbf{C1}$).
$\langle 3 \rangle 3$. By construction, we either obtain a run $r' = r'_1 \xrightarrow{1} r'_M = \mathsf{error}$ such that $M \leq N$, or a run $r' = r'_1 \xrightarrow{1} r'_N$ such that $r'_N \neq \mathsf{error}$ and $\forall 1 \neq j \leq N.\ r'_j = (\overline{\sigma}_{r_j}, \epsilon_{r_j}[\beta/\alpha])$ where $\overline{\sigma}_{r_j}|_V = \sigma_{r_j}$.
$\langle 3 \rangle 4$. Then either $(\overline{\sigma}_{r_1}|_V) \in Bad(P_{i+1}, \exists X.\mathcal{I})$ or $(\overline{\sigma}_{r_1}|_V, \overline{\sigma}_{r_N}|_V) \in Good(P_{i+1}, \exists X.\mathcal{I})$.
$\langle 3 \rangle 5$. Q.E.D.

CASE: Rule HIDE-VARIABLE: Remove the existing variable v from the program, and replace the invariant $\mathcal{I}$ with $\exists v.\, \mathcal{I}$. Replace every action α with β whenever $(P, \mathcal{I}) \vdash \alpha \preceq_{-v} \beta$, which holds if the following are both valid:

$$\mathbf{H1}.\quad (\exists v.\, \mathcal{I} \land \neg\phi_\alpha) \;\Rightarrow\; \neg\phi_\beta \qquad \mathbf{H2}.\quad (\exists v, v'.\, \mathcal{I} \land \tau_\alpha) \;\Rightarrow\; (\neg\phi_\beta \lor \tau_\beta)$$

ASSUME: $\mathcal{I}_{i+1} = \exists v.\mathcal{I}_i$, by proof rule.
ASSUME: $X = \{v\}$, by proof rule. Use v instead of X in the formulas.
ASSUME: $V = Var_{prog_{i+1}}$, since $Var_{P_i} = Var_{P_{i+1}} \cup \{v\}$.
ASSUME: $\exists X.\mathcal{I}_i = \exists v.\mathcal{I}_i = \exists v.\mathcal{I}_{i+1}$. Call this invariant $\mathcal{I}$ in the following. Note that $\mathcal{I}$ does not refer to v.
PROVE: **C1**.
$\langle 3 \rangle 1$. Take a run $r = r_1 \xrightarrow{1} r_N = \mathsf{error}$ of P_1 such that $r_1 \vDash \mathcal{I}$.
$\langle 3 \rangle 2$. Construct the run r' inductively.
CASE: Base case
$\langle 5 \rangle 1$. Take $r'_1 = (\sigma_{r_1})|_V, \epsilon_{r_1}[\beta/\alpha]$.
$\langle 5 \rangle 2$. Since $\sigma_{r_1} \vDash \mathcal{I}$ and $\mathcal{I}$ does not refer to v, $\sigma_{r_1}|_V \vDash \mathcal{I}$ holds. Thus $r'_1 \vDash \mathcal{I}$.
CASE: Inductive case
ASSUME: $r'_1 = r'_1 \xrightarrow{1'} r'_k$ such that $\forall 1 \leq j \leq k. r'_j = (\sigma_{r_j}|_V, \epsilon_{r_j}[\beta/\alpha])$.
$\langle 5 \rangle 1$. Take the transition $r_k \xrightarrow{(t,\alpha)} r_{k+1}$. There are two cases $r_{k+1} = \mathsf{error}$ or not.
ASSUME: $r_k \xrightarrow{(t,\alpha)} \mathsf{error}$.

$\langle 5 \rangle 2$. By condition $\mathbf{H1}$, if $\sigma_{r_k} \models \neg\phi_\alpha[t]$ for the value of $\sigma_{r_j}(v)$, then $\sigma_{r_k}|_V \models \neg\phi_\beta[t]$. Thus $r'_k \xrightarrow{(t,\alpha)} \mathsf{error}$.

ASSUME: $r_k \xrightarrow{(t,\alpha)} r_{k+1}$ and $r_{k+1} \neq \mathsf{error}$.

$\langle 5 \rangle 3$. By condition $\mathbf{H2}$, if $\sigma_{r_k} \models \phi_\alpha[t]$ and $\tau_\alpha(\sigma_{r_k}, \sigma_{k+1})$ is true for values of $\sigma_{r_k}(v)$ and $\sigma_{r_{k+1}}(v)$, then, either $\sigma_{r_k}|_V \models \neg\phi_\beta[t]$ or $\tau_\beta(\sigma_{r_k}|_V, \sigma_{r_{k+1}}|_V)$ is true. In the former case, we reach a violation, satisfying $\mathbf{C1}$, or obtain the run $r' = r'_1 \longrightarrow \mathbf{l}'(t,\beta) r'_{k+1}$ where $r'_{k+1} = (\sigma_{r_{k+1}}|_V, \epsilon_{r_{k+1}}[\beta/\alpha])$.

$\langle 3 \rangle 3$. By the above construction, we obtain a run $r' = r'_1 \xrightarrow{\mathbf{1}} r'_M = \mathsf{error}$ such that $M \leq N$. The case $r_N = \mathsf{error}$ ensures that the above construction reaches a violation when translating the transition $r_{N-1} \xrightarrow{(t,\alpha)} r_N = \mathsf{error}$ at the latest.

$\langle 3 \rangle 4$. Then $(\sigma_{r'_1}) = (\sigma_{r_1}|_V) \in \mathit{Bad}(P_{i+1}, \mathcal{I})$.

$\langle 3 \rangle 5$. Q.E.D.

PROVE: $\mathbf{C2}$.

$\langle 3 \rangle 1$. Take a run $r = r_1 \xrightarrow{\mathbf{1}} r_N$ where $r_N \neq \mathsf{error}$ of P_1 such that $r_1 \models \exists X.\mathcal{I}$.

$\langle 3 \rangle 2$. We construct the run r' inductively, similarly to the above case ($\mathbf{C1}$).

$\langle 3 \rangle 3$. By construction, we either obtain a run $r' = r'_1 \xrightarrow{\mathbf{1}} r'_M = \mathsf{error}$ such that $M \leq N$, or a run $r' = r'_1 \xrightarrow{\mathbf{1}} r'_N$ such that $r'_N \neq \mathsf{error}$ and $\forall 1 \neq j \leq N.\ r'_j = (\sigma_{r_j}|_V, \epsilon_{r_j}[\beta/\alpha])$.

$\langle 3 \rangle 4$. Then either $(\sigma_{r'_1}) = (\sigma_{r_1}|_V) \in \mathit{Bad}(P_{i+1}, \exists X.\mathcal{I})$ or $(\sigma_{r'_1}, \sigma_{r'_N}) = (\sigma_{r_1}|_V, \sigma_{r_N}|_V) \in \mathit{Good}(P_{i+1}, \exists X.\mathcal{I})$.

$\langle 3 \rangle 5$. Q.E.D.

$\langle 2 \rangle 1$. Q.E.D.

C. Soundness of the QED steps

Notation. In this section, we use the following notation in addition to the existing ones to express executions for clarity.

- We will eliminate from some executions the steps (SEQUENTIAL, JOIN-FIRST, JOIN-SECOND) whose operation is to remove skip statements. For example, the execution $(\sigma_1, \alpha_1; \alpha_2) \longrightarrow (\sigma_2, \mathsf{skip}; \alpha_2) \longrightarrow (\sigma_2, \alpha_2)$ will be expressed as $(\sigma_1, \alpha_1; \alpha_2) \longrightarrow (\sigma_2, \alpha_2)$.
- We will sometimes omit the statement component of the program state if its content is not necessary, and will only show the store component. We do this only for states whose statement component is not error. For example, the execution $(\sigma_1, \alpha_1 \Box \alpha_2) \xrightarrow{\alpha_1} (\sigma_2, \mathsf{skip})$ will be shown as $\sigma_1 \xrightarrow{\alpha_1} \sigma_2$ if the initial statement is not important but the action evaluated is.
- We sometimes make use of the following context with one hole:

$$C \quad ::= \quad [\cdot] \mid c\ ;\ C \mid C\ ;\ c \mid c \Box C \mid C \Box c \mid C \parallel c \mid c \parallel C \mid C^{\circlearrowleft}$$

LEMMA 7 (Preservation). *Let $P_1, \mathcal{I}_1 \dashrightarrow P_2, \mathcal{I}_2$ be a proof step. Let $\mathrm{V} = P_1.\mathit{Var}$ and $\mathrm{X} = P_2.\mathit{Var} \backslash P_1.\mathit{Var}$. Then the following holds:*

1. *For each $(\sigma_1, \sigma_2) \in \mathit{Bad}(P_1, \exists X.\, \mathcal{I}_2)$:*
   $\exists \sigma_3.\ (\sigma_1, \sigma_3) \in \mathit{Bad}|_{\mathrm{V}}(P_2, \mathcal{I}_2)$.
2. *For each $(\sigma_1, \sigma_2) \in \mathit{Good}(P_1, \exists X.\, \mathcal{I}_2)$:*
   (a) $\exists \sigma_3.\ (\sigma_1, \sigma_3) \in \mathit{Bad}|_{\mathrm{V}}(P_2, \mathcal{I}_2)$, or
   (b) $(\sigma_1, \sigma_2) \in \mathit{Good}|_{\mathrm{V}}(P_2, \mathcal{I}_2))$

PROOF SKETCH:We will do a case split on the kind of rule applied during the proof step $P_1, \mathcal{I}_1 \dashrightarrow P_2, \mathcal{I}_2$. While proving each rule, we will first assume an execution $(\sigma_1, t_{main} : \mathit{Main}_1) \longrightarrow^* (\sigma_2, d)$, where $d \in \{\mathsf{skip}, \mathsf{error}\}$, of P_1 and show that we can derive a *witness* execution of P_2, which is either of the form $(\sigma_1, t_{main} : \mathit{Main}_2) \longrightarrow^* (\sigma_2, d)$ or of the form $(\sigma_1, t_{main} : \mathit{Main}_2) \longrightarrow^* (\sigma_3, \mathsf{error})$, for some σ_3. Existence of a witness execution of P_2 implies the claims of the lemma: If $d = \mathsf{error}$ then both witness executions satisfy Condition *1* of the lemma by showing that P_2 goes wrong from every state from which P_1 goes wrong. If $d \neq \mathsf{error}$ then either the first witness execution satisfies Condition *2(a)*, or the second witness execution satisfies Condition *2(b)* of the lemma.

AUX-ANNOTATE

$$\frac{\begin{array}{ll} \mathit{Atoms}(P) = \{\phi^1 \triangleright \tau_1^1, \ldots, \phi^n \triangleright \tau_1^n\} & \phi^1 \triangleright \tau_2^1 \vdash \mathcal{I} \quad \ldots \quad \phi^n \triangleright \tau_2^n \vdash \mathcal{I} \\ a \notin \mathit{Var} & \models (\phi^1 \wedge \tau_1^1) \Rightarrow \forall a. \exists a'.\, \tau_2^1 \quad \ldots \quad \models (\phi^n \wedge \tau_1^n) \Rightarrow \forall a. \exists a'.\, \tau_2^n \end{array}}{P, \mathcal{I} \dashrightarrow P[\mathit{Var} \mapsto \mathit{Var} \cup \{a\}, \phi^1 \triangleright \tau_1^1 \mapsto \phi^1 \triangleright \tau_2^1, \ldots, \phi^n \triangleright \tau_1^n \mapsto \phi^n \triangleright \tau_2^n], \mathcal{I}}$$

CASE: AUX-ANNOTATE

PROOF SKETCH:We prove that for any execution $(\sigma_1, t_{main} : \mathit{Main}_1) \longrightarrow^* (\sigma_2, d)$ of P_1, where d is an arbitrary dynamic statement, there exists an execution $(\overline{\sigma}_1, t_{main} : \mathit{Main}_2) \longrightarrow^* (\overline{\sigma}_2, \overline{d})$ of P_2, where $\overline{d}$ is d with actions in $\mathit{Atoms}(d)$ replaced with the corresponding ones indicated by the rule, and $\overline{\sigma}_1|_{P_1.\mathit{Var}} = \sigma_1$ and $\overline{\sigma}_2|_{P_1.\mathit{Var}} = \sigma_2$.

⟨3⟩1. Since the rule does not change the program invariant, let $\mathcal{I}_1 = \mathcal{I}_2 = \mathcal{I}$.

PROVE: Let $E : (\sigma^1, d^1) \xrightarrow{\phi^1 \triangleright \tau_1^1} (\sigma^2, d^2) \xrightarrow{\phi^2 \triangleright \tau_1^2} (\sigma^3, d^3) \cdots \xrightarrow{\phi^n \triangleright \tau_1^n} (\sigma^{n+1}, d^{n+1})$ be an execution of P_1 such that $d^1 = t_{main} : P_1.\mathit{Main}$. There exists an execution $(\overline{\sigma}^1, \overline{d}^1) \xrightarrow{\phi^1 \triangleright \tau_2^1} (\overline{\sigma}^2, \overline{d}^2) \xrightarrow{\phi^2 \triangleright \tau_2^2} (\overline{\sigma}^3, \overline{d}^3) \cdots \xrightarrow{\phi^n \triangleright \tau_2^n} (\overline{\sigma}^{n+1}, \overline{d}^{n+1})$ of P_2 such that $\forall i \in [1..n{+}1].\ \overline{\sigma}^i|_{P_1.\mathit{Var}} = \sigma^i$, and $\overline{d}^i$ is d^i after the actions in $\mathit{Atoms}(d_1^i)$ are replaced with the corresponding ones indicated by the rule.

⟨4⟩1. Proof by induction on the length of the execution (the number of transitions in it) of P_1.

CASE: Base case: If $n = 0$ then there is no transition in the program. In this case we can extend σ^1, by having an arbitrary assignment to a, to a store $\overline{\sigma}^1$ such that $\overline{\sigma}^1|_{P_1.\mathit{Var}} = \sigma^1$. Since the invariant does not refer to the variable a, $\overline{\sigma}^1 \models \mathcal{I}$.

CASE: Inductive case: Assume that the inductive hypothesis holds for executions of length n. We will prove the claim for executions of length $n + 1$.

⟨5⟩1. Assume that the execution of P_1 is of the form $E : (\sigma^1, d^1) \longrightarrow \cdots \longrightarrow (\sigma^{n+1}, d^{n+1}) \xrightarrow{\phi^{n+1} \triangleright \tau_1^{n+1}} (\sigma^{n+2}, d^{n+2})$. Assume also that the inductive hypothesis holds for the prefixes of the execution with $\leq n$ transitions. Thus for the prefix of there exists an execution $E' : (\overline{\sigma}^1, \overline{d}^1) \longrightarrow \cdots \longrightarrow (\overline{\sigma}^{n+1}, \overline{d}^{n+1})$ satisfying the claim.

⟨5⟩2. Consider the last transition of E: $(\sigma^{n+1}, d^{n+1}) \xrightarrow{\phi^{n+1} \triangleright \tau_1^{n+1}} (\sigma^{n+2}, d^{n+2})$. Let $\phi^{n+1} \triangleright \tau_2^{n+1}$ be the action that is replaced by $\phi^{n+1} \triangleright \tau_1^{n+1}$ by the rule.

⟨5⟩3. $(\sigma^{n+1}, \sigma^{n+2}) \models \tau_1^{n+1}$ holds by ATOMIC. By the premise $\tau_1^{n+1} \Rightarrow \forall a. \exists a'.\ \tau_2^{n+1}$. Thus we can write $(\sigma^{n+1}, \sigma^{n+2}) \models \forall a.\ \exists a'.\ \tau_2^{n+1}$.

⟨5⟩4. Let $\overline{\sigma}^{n+2}|_{P_1.\mathit{Var}} = \sigma^{n+2}$ such that $(\overline{\sigma}^{n+1}, \overline{\sigma}^{n+2}) \models \tau_2^{n+1}$ holds. We choose the valuation of a in $\overline{\sigma}^{n+2}$ by selecting the value of a' that make the formula $\forall a.\ \exists a'.\ \tau_2^{n+1}$ true given the value of a in $\overline{\sigma}^{n+1}$. Note that the formula implies that we can always choose a value for a' for any value of a.

⟨5⟩5. $\sigma^{n+1} \models \phi^{n+1}$ by ATOMIC, thus $\overline{\sigma}^{n+1} \models \phi^{n+1}$. By ⟨5⟩4, there exists a transition $\overline{\sigma}^{i+1} \xrightarrow{\phi^{n+1} \triangleright \tau_2^{n+1}} (\overline{\sigma}^{n+2}, \mathsf{skip})$.

⟨5⟩6. By ⟨5⟩5, extending the execution E' of P_2 with the transition $(\overline{\sigma}^{n+1}, \overline{d}^{n+1}) \xrightarrow{\phi^{n+1} \triangleright \tau_2^{n+1}} (\overline{\sigma}^{n+2}, \overline{d}^{n+1})$ yields the desired result.

⟨4⟩2. Q.E.D.

ASSUME: While proving the remaining rules, we will consider $P_1.Var = P_2.Var$, and thus $\text{X} = \emptyset$. In addition, $Good|_{\text{V}}(\cdot) = Good(\cdot)$ and $Bad|_{\text{V}}(\cdot) = Bad(\cdot)$.

$$\frac{\vDash \mathcal{I}_2 \Rightarrow \mathcal{I}_1 \qquad P \vdash \mathcal{I}_2}{P, \mathcal{I}_1 \dashrightarrow P, \mathcal{I}_2}\ \text{INVARIANT}$$

CASE: INVARIANT

⟨4⟩1. Since the strengthened invariant $\mathcal{I}_2$ is used in the lemma, and the rule does not modify the program at all, it holds trivially that $Good(P_1, \mathcal{I}_2) = Good(P_2, \mathcal{I}_2)$ and $Bad(P_1, \mathcal{I}_2) = Bad(P_2, \mathcal{I}_2)$.

⟨4⟩2. Note that the premise $\mathcal{I}_2 \Rightarrow \mathcal{I}_1$ is not required for proving this lemma, but it will be essential when proving Theorem 1.

⟨4⟩3. Q.E.D.

ASSUME: The remaining rules proved below do not modify the invariant, so we will consider $\mathcal{I} = \mathcal{I}_1 = \mathcal{I}_2$ from this point on.

$$\frac{\mathcal{I} \vdash \alpha_1 \preceq \alpha_2 \qquad \alpha_2 \vdash \mathcal{I}}{P, \mathcal{I} \dashrightarrow P[\alpha_1 \mapsto \alpha_2], \mathcal{I}}\ \text{SIMULATE}$$

CASE: SIMULATE

PROOF SKETCH:We prove that replacing transitions of α_1 in an execution of P_1 with a transition of α_2 either causes the new execution to go wrong, or leads the new execution to the same final to which the original one goes. Iterative application of this yields a witness execution of P_2 that goes wrong earlier or ends at the same program state.

PROVE: Let $E : (\sigma_1, d^1) \longrightarrow \cdots \longrightarrow (\sigma_2, d^2)$ be an execution of P_1 where $d^1 = P_1.Main$. There exists either an execution $E' : (\sigma_1, \overline{d}^1) \longrightarrow \cdots \longrightarrow (\sigma_2, \overline{d}^2)$ or $E' : (\sigma_1, \overline{d}^1) \longrightarrow \cdots \longrightarrow (\sigma_3, \mathsf{error})$ of P_2 such that each $\overline{d}^i$ for some i is d^i except for the occurrence of the action α_1 is replaced with α_2 by the rule.

⟨4⟩1. Proof by induction on the number of transitions of the execution of P_1.

CASE: Base case: If $n = 0$ then there is no transition in the program. In this case the same execution is a witness to the claim.

CASE: Inductive case: Assume that the inductive hypothesis holds for executions of length n. We will prove the claim for the executions of length $n + 1$.

⟨5⟩1. Consider the following execution: $E : (\sigma^1, d^1) \longrightarrow \cdots \xrightarrow{\alpha^n} (\sigma^{n+1}, d^{n+1}) \xrightarrow{\alpha^{n+1}} (\sigma^{n+2}, d^{n+2})$. Assume also that the inductive hypothesis holds for the prefixes of the execution with $\leq n$ transitions. Thus there exists either an execution $E'_1 : (\sigma_1, \overline{d}^1) \longrightarrow \cdots \longrightarrow (\overline{\sigma}^{n+1}, \mathsf{error})$ or an execution $E'_2 : (\sigma_1, \overline{d}^1) \longrightarrow \cdots \longrightarrow (\sigma^{n+1}, \overline{d}^{n+1})$. If the former holds, execution E'_1 P_2 is also a witness for E to the claim. Assume the latter in the following.

⟨5⟩2. Consider the last transition of E: $(\sigma^{n+1}, d^{n+1}) \xrightarrow{\alpha^{n+1}} (\sigma^{n+2}, d^{n+2})$.

⟨5⟩3. If $\alpha^{n+1} \neq \alpha_1$, then the execution E'_2 of P_2 can be extended by the transition $(\sigma^{n+1}, \overline{d}^{n+1}) \xrightarrow{\alpha^{n+1}} (\sigma^{n+2}, \overline{d}^{n+2})$, which yields a witness execution.

⟨5⟩4. If $\alpha^{n+1} = \alpha_1$, then existence of the transition $(\sigma^{n+1}, d^{n+1}) \xrightarrow{\alpha_1} (\sigma^{n+2}, d^{n+2})$ indicates the use of the transition $(\sigma^{n+1}, \alpha_1) \longrightarrow (\sigma^{n+2}, d)$ where $d \in \{\mathsf{skip}, \mathsf{error}\}$ in the derivation of E. This, by the definition of $\preceq$, implies that there exists either a transition $(\sigma^{n+1}, \alpha_2) \longrightarrow (\sigma^{n+2}, d)$ or the transition $(\sigma^{n+1}\alpha_2) \longrightarrow (\sigma^{n+1}, \mathsf{error})$. We use either of these transitions to extend the execution of E'_2 of P_2 with either the transition $(\sigma^{n+1}, \overline{d}^{n+1}) \xrightarrow{\alpha_2} (\sigma^{n+2}, \overline{d}^{n+2})$ or the transition $(\sigma^{n+1}, \overline{d}^{n+1}) \xrightarrow{\alpha_2} (\sigma^{n+1}, \mathsf{error})$ yields a witness execution for E to the claim.

⟨5⟩5. Q.E.D.

$$\frac{\vDash \mathcal{I} \Rightarrow \phi}{P, \mathcal{I} \dashrightarrow P[\phi \rhd \tau \mapsto \mathsf{true} \rhd \tau], \mathcal{I}}\ \text{RELAX*}$$

CASE: RELAX

⟨4⟩1. By having $\alpha_1 = \phi \rhd \tau$ and $\alpha_2 = \mathsf{true} \rhd \tau, \mathcal{I} \Rightarrow \phi$ we can apply Lemma 8 to show that $\alpha_1 \preceq \alpha_2$, and then apply SIMULATE to replace actions.

⟨4⟩2. Q.E.D.

$$\frac{P, \mathcal{I} \vdash \alpha_1 : \mathbb{R} \quad \text{or} \quad P, \mathcal{I} \vdash \alpha_2 : \mathbb{L}}{P, \mathcal{I} \dashrightarrow P[\alpha_1 ; \alpha_2 \mapsto \alpha_1 \circ \alpha_2], \mathcal{I}}\ \text{REDUCE-SEQUENTIAL}$$

CASE: REDUCE-SEQUENTIAL

PROOF SKETCH:Given an execution $E : (\sigma_1, t_{main} : Main_1) \longrightarrow^* (\sigma_2, d)$ of P_1, we follow a 3-phase approach to derive a witness execution of P_2. First, we move right-mover actions to the right and left-mover actions to the left to sort the transitions in the execution such that the transitions that belong to $\alpha_1; \alpha_2$ become adjacent to each other and the relative order of other actions are preserved. While doing this, we may generate executions that either goes wrong earlier or go to the same final state as the original execution goes. Second, if we get an execution that goes wrong, we eliminate from the execution transitions of unmatched α_1 actions (with corresponding α_2 transitions not executed in the execution yet). Third, we replace each transition sequence of $\alpha_1; \alpha_2$ with $\alpha_1 \circ \alpha_2$, which yields a witness execution of P_2 to the claims of the lemma.

⟨3⟩1. **First phase:** Consider a given execution $E : (\sigma_1, t_{main} : Main_1) \longrightarrow^* (\sigma_2, d)$ of P_1. In this phase, we sort the actions in the execution by moving mover actions, which at the end results in an execution in which transitions of every $\alpha_1; \alpha_2$ pair appear adjacent.

⟨3⟩2. We first define a commit action for each $\alpha_1; \alpha_2$ pair. If either α_1 is right-mover and if α_2 also exists in the execution α_2 is the commit action, otherwise α_1 is the commit action. If α_2 is left-mover then α_1 is the commit action. If both cases hold, the commit action is arbitrarily chosen to be either α_1 or α_2.

⟨3⟩3. We define $Sorted(E)$ as to be an execution containing the same set of actions in E and specifying a total ordering between actions in E in the following way: Let $E : (\sigma^1, d^1) \xrightarrow{\alpha^1} (\sigma^2, d^2) \xrightarrow{\alpha^2}\longrightarrow \cdots \xrightarrow{\alpha^n} (\sigma^n, d^n)$. Let $ToMerge(E)$ be the entire set of transitions of α_1, α_2. Let $commit(E, \alpha)$ be the commit action for the pair $\alpha_1; \alpha_2$ where $\alpha = \{\alpha_1, \alpha_2\}$. Let $iscommit(E, \alpha)$ hold if α is a commit action for a $\alpha_1; \alpha_2$ pair in E. Define a total order $\ll_E$ between actions in E and a total order $\ll_{Sorted(E)}$ between actions in $Sorted(E)$.

$\alpha^i \ll_{Sorted(E)} \alpha^j$ iff
1) $\alpha^i \ll_E \alpha^j$ and $\alpha^i \notin ToMerge(E)$ and $\alpha^j \notin ToMerge(E)$,
2) $\alpha^i \ll_E \alpha^j$ and $iscommit(E, \alpha^i)$ and $\alpha^j \notin ToMerge(E)$,
3) $\alpha^i \ll_E \alpha^j$ and $\alpha^i \notin ToMerge(E)$ and $iscommit(E, \alpha^j)$,
4) $\alpha^i \ll_E \alpha^j$ and $iscommit(E, \alpha^i)$ and $iscommit(E, \alpha^j)$,
5) $\alpha^i \in ToMerge(E)$, $\alpha^j \notin ToMerge(E)$ and $commit(E, \alpha^i) \ll_{Sorted(E)} \alpha^j$,
6) $\alpha^i \notin ToMerge(E)$, $\alpha^j \in ToMerge(E)$ and $\alpha^i \ll_{Sorted(E)} commit(E, \alpha^j)$,
7) $\alpha^i, \alpha^j \in ToMerge(E)$ and $commit(E, \alpha^i) \ll_{Sorted(E)} commit(E, \alpha^j)$.

⟨3⟩4. Let $M(\cdot)$ be a measure function over executions such that $M(E)$ is the total number of inversions, i.e. $\alpha^i \ll_{Sorted(E)} \alpha^j$ but $\alpha^i \ll_E \alpha^j$. Each Each swap while *bubble-sorting* the execution E to get $Sorted(E)$ is done between a transition of a right-mover α_1 or a left-mover α_2 with another action α^i that involves an inversion. As we will show below that each swap either shortens the execution by causing it to go wrong earlier, or generates a new execution E' with $M(E') < M(E)$. Thus the fact that $M(E)$ for each execution E is finite implies that this phase terminates outputting a sorted execution (possibly shorter than original one). In the following, we will show the property of each swap that ensures this termination guarantee and the final form of the execution output from this phase.

PROVE: Let $E : (\sigma_1, t_{main} : Main_1) \longrightarrow^* (\sigma_2, d)$ be the current execution before a swap of α_1 or α_2 with another action α^i performed by a different thread. The swap generates either execution $E' : (\sigma_1, t_{main} : Main_1) \longrightarrow^* (\sigma_3, \mathsf{error})$ for some σ_3 (possibly shorter than E), or an execution $E' : (\sigma_1, t_{main} : Main_1) \longrightarrow^* (\sigma_2, d)$ such that $M(E') < M(E)$.

⟨4⟩1. Let current the execution be $E : (\sigma_1, t_{main} : Main_1) \longrightarrow^* (\sigma_2, d)$.

⟨4⟩2. There are two cases: Either $P, \mathcal{I} \vdash \alpha_1 : \mathbb{R}$ and α_1 moves to the right, or $P, \mathcal{I} \vdash \alpha_2 : \mathbb{L}$ and α_2 moves to the left.

CASE: $P_1, \mathcal{I} \vdash \alpha_1 : \mathbb{R}$

⟨5⟩1. Assume that the execution is of the form: $(\sigma_1, t_{main} : Main_1) \longrightarrow \cdots \longrightarrow (\sigma^{i-1}, t : C[\alpha_1; \alpha_2]) \xrightarrow{\alpha_1} (\sigma^i) \xrightarrow{\alpha^i} (\sigma^{i+1}) \longrightarrow \cdots \longrightarrow (\sigma_2, d)$. By the definition of right-movers, we can move the evaluation of α_1 to the right of α^i to obtain another execution of P in the following way: The transition sequence $(\sigma^{i-1}) \xrightarrow{\alpha_1} (\sigma^i) \xrightarrow{\alpha^i} (\sigma^{i+1})$ can be replaced with either the transition $(\sigma^{i-1}) \xrightarrow{\alpha^i} (\sigma^{i-1}, \mathsf{error})$, or the transition sequence $(\sigma^{i-1}) \xrightarrow{\alpha^i} (\overline{\sigma}) \xrightarrow{\alpha_1} (\overline{\sigma}, \mathsf{error})$, or the transition sequence $(\sigma^{i-1}) \xrightarrow{\alpha^i} (\overline{\sigma}) \xrightarrow{\alpha_1} (\sigma^{i+1})$ for some $\overline{\sigma}$.

⟨5⟩2. In the first case, we obtain a shorter execution E' that goes wrong: $E' : (\sigma_1, t_{main} : Main_2) \longrightarrow \cdots \longrightarrow (\sigma^{i-1}, t : C[\alpha_1; \alpha_2]) \xrightarrow{\alpha^i} (\sigma^{i-1}, \mathsf{error})$

⟨5⟩3. If the second case happens, we obtain an execution E' that goes wrong and $M(E') < M(E)$: $E' : (\sigma_1, t_{main} : Main_2) \longrightarrow \cdots \longrightarrow (\sigma^{i-1}, t : C[\alpha_1; \alpha_2]) \xrightarrow{\alpha^i} (\overline{\sigma}) \xrightarrow{\alpha_1} (\overline{\sigma}, \mathsf{error})$.

⟨5⟩4. If the third case happens, we obtain an execution that goes to the same program state as the previous one goes: $(\sigma_1, t_{main} : Main_1) \longrightarrow \cdots \longrightarrow (\sigma^{i-1}, t : C[\alpha_1; \alpha_2]) \xrightarrow{\alpha^i} (\overline{\sigma}) \xrightarrow{\alpha_1} (\sigma^{i+1}) \longrightarrow \cdots \longrightarrow (\sigma_2, d)$, by executing the rest of the execution from (σ^{i+1}) as in the previous execution. In this case $M(E') < M(E)$.

CASE: $P_1, \mathcal{I} \vdash \alpha_2 : \mathbb{L}$

⟨5⟩1. Assume that the execution be of the form: $E : (\sigma_1, t_{main} : Main_1) \longrightarrow \cdots \xrightarrow{\alpha_1} \cdots \longrightarrow (\sigma^{i-1}) \xrightarrow{\alpha^i} (\sigma^i) \xrightarrow{\alpha_2} (\sigma^{i+1}) \longrightarrow \cdots \longrightarrow (\sigma_2, d)$. By the definition of left-movers, we can move the evaluation of α_2 to the left of α^i to obtain another execution of P in the following way: The transition sequence $(\sigma^{i-1}) \xrightarrow{\alpha^i} (\sigma^i) \xrightarrow{\alpha_2} (\sigma^{i+1})$ can be replaced with either the transition $(\sigma^{i-1}) \xrightarrow{\alpha_2} (\sigma^{i-1}, \mathsf{error})$, or the transition sequence $(\sigma^{i-1}) \xrightarrow{\alpha_2} (\overline{\sigma}) \xrightarrow{\alpha^i} (\overline{\sigma}, \mathsf{error})$ for some $\overline{\sigma}$ or the transition sequence $(\sigma^{i-1}) \xrightarrow{\alpha_2} (\overline{\sigma}) \xrightarrow{\alpha^i} (\sigma^{i+1})$.

⟨5⟩2. In the first case, we obtain an execution that goes wrong before executing α^i: $E : (\sigma_1, t_{main} : Main_1) \longrightarrow \cdots \xrightarrow{\alpha_1} \cdots \longrightarrow (\sigma^{i-1}) \xrightarrow{\alpha_2} (\sigma^i, \mathsf{error})$. Thus E' is shorter than E.

⟨5⟩3. In the second case, we obtain an execution that goes wrong: $(\sigma_1, t_{main} : Main_1) \longrightarrow \cdots \xrightarrow{\alpha_1} \cdots \longrightarrow (\sigma^{i-1}) \xrightarrow{\alpha_2} (\overline{\sigma}) \xrightarrow{\alpha^i} (\overline{\sigma}, \mathsf{error})$. In this case $M(E') < M(E)$.

⟨5⟩4. If the third case happens, we obtain an execution that goes to the same program state as the previous one goes: $(\sigma_1, t_{main} : Main_1) \longrightarrow \cdots \xrightarrow{\alpha_1} \cdots \longrightarrow (\sigma^{i-1}) \xrightarrow{\alpha_2} (\overline{\sigma}) \xrightarrow{\alpha^i} (\sigma^{i+1}) \longrightarrow \cdots \longrightarrow (\sigma_2, d)$, by executing the rest of the execution from (σ^{i+1}) as in the previous execution. In this case $M(E') < M(E)$.

⟨4⟩3. Q.E.D.

⟨3⟩5. We perform the above iteration until we obtain either an execution $E' : (\sigma_1, t_{main} : Main_1) \longrightarrow^* (\sigma_3, \mathsf{error})$ for some σ_3, or an execution $E' : (\sigma_1, t_{main} : Main_1) \longrightarrow^* (\sigma_2, d)$. In both cases $M(E') = 0$.

$\langle 3\rangle 6$. **Second phase:** In this phase, we eliminate transitions of right-mover α_1 (except for a possible last right-mover α_1) that have no matching transition of α_2 in the execution. This situation is possible if the current execution goes wrong. We perform this elimination by generating a shorter execution that goes wrong without executing these unmatched transitions: We move each transitions of α_1 to the right until we obtain an execution that goes wrong without executing that transition of α_1.

$\langle 4\rangle 1$. To justify this, consider a right-mover α_1. We move this action to the right in a similar way described in the first phase. A movement of α_1 either yields a new execution that either goes wrong earlier than the current one or ends in the same final state as the current one ends. Suppose that we moved α_1 until it gets executed before the last action of the execution that goes wrong. Let the following be the last two transitions of the execution: $(\sigma^k, d^k) \xrightarrow{\alpha_1} (\sigma^l, d^l) \xrightarrow{\alpha^i} (\sigma^n, \mathsf{error})$. By definition of right-movers, there exists a transition $(\sigma^k, d^k) \xrightarrow{\alpha^i} (\sigma^k, \mathsf{error})$. With the same prefix of the current one, this yields an execution that goes wrong and does not include the transition of α_1.

$\langle 4\rangle 2$. Note that this process does not prevent the situation that there is one last transition of a right-mover α_1 at the end, which goes wrong. This happens if the last action of the execution output from the first phase, which goes wrong, is an unmatched α_1. We will deal with that transition in the next phase.

$\langle 3\rangle 7$. **Third phase:** In this phase we replace each transition sequence of a $\alpha_1; \alpha_2$ pair in the execution with a single transition of $\alpha_1 \circ \alpha_2$.

$\langle 4\rangle 1$. By definition of $\circ$, We can replace each the transition sequence $(\sigma^1, t : C[\alpha_1; \alpha_2]) \xrightarrow{\alpha_1} (\sigma^2, t : C[\alpha_2]) \xrightarrow{\alpha_2} (\sigma^3, t : C[\mathsf{skip}])$ with the transition $(\sigma^1, t : C[\alpha_1 \circ \alpha_2]) \xrightarrow{\alpha_1 \circ \alpha_2} (\sigma^3, t : C[\mathsf{skip}])$. In addition we can replace $(\sigma^1, t : C[\alpha_1; \alpha_2]) \xrightarrow{\alpha_1} (\sigma^1, \mathsf{error})$ and $(\sigma^1, t : C[\alpha_1; \alpha_2]) \xrightarrow{\alpha_1} (\sigma^2, t : C[\alpha_2]) \xrightarrow{\alpha_2} (\sigma^2, \mathsf{error})$ with $(\sigma^1, t : C[\alpha_1 \circ \alpha_2]) \xrightarrow{\alpha_1 \circ \alpha_2} (\sigma^1, \mathsf{error})$. In all cases, the resulting execution satisfies the claims of the lemma.

$\langle 4\rangle 2$. Finally, assume that the last transition of the execution (if it goes wrong) is $(\sigma^k, d^k) \xrightarrow{\alpha_1} (\sigma_k, \mathsf{error})$, which belongs to an unmatched right mover α_1. We replace this transition with the transition $(\sigma^k, d^k) \xrightarrow{\alpha_1 \circ \alpha_2} (\sigma_k, \mathsf{error})$. This is valid by definition of $\circ$.

$\langle 3\rangle 8$. Q.E.D.

$$\frac{\text{REDUCE-CHOICE}}{P, \mathcal{I} \dashrightarrow P[\alpha_1 \Box \alpha_2 \mapsto \alpha_1 \oplus \alpha_2], \mathcal{I}}$$

CASE: REDUCE-CHOICE

PROOF SKETCH:We handle all evaluations of $\alpha_1 \Box \alpha_2$ in an iterative approach; in each iteration we take an execution and replace evaluations of the selected action with the evaluation of $\alpha_1 \oplus \alpha_2$, and we show that result of an iteration does not break the reasoning in other iterations. By the definition of $\oplus$, the new transition either goes wrong or can go to the same post-store that the selected action goes in the original execution. Thus, the witness execution either goes wrong or ends at the same final state the original execution ends.

PROVE: Let $E : (\sigma_1, d^1) \longrightarrow \cdots \longrightarrow (\sigma_2, d^2)$ be an execution of P_1 where $d^1 = P_1.\mathit{Main}$. There exists an execution either an execution $(\sigma_1, \overline{d}^1) \longrightarrow \cdots \longrightarrow (\sigma_2, \overline{d}^2)$ or $(\sigma_1, \overline{d}^1) \longrightarrow \cdots \longrightarrow (\sigma_3, \mathsf{error})$ of P_2 such that each $\overline{d}^i$ for some i is d^i except for the occurrence of the statement $\alpha_1 \Box \alpha_2$, or an already selected component of it is replaced with $\alpha_1 \oplus \alpha_2$ by the rule.

$\langle 4\rangle 1$. Proof by induction on the number of transitions of the execution of P_1.

CASE: Base case: If $n = 0$ then there is no transition in the program. In this case the same execution is a witness to the claim.

CASE: Inductive case: Assume that the inductive hypothesis holds for execution of length n. We will prove the claim for the executions of length $n + 1$.

$\langle 5\rangle 1$. Assume that the execution is of the form $E : (\sigma^1, d^1) \longrightarrow \cdots \xrightarrow{\alpha^n} (\sigma^{n+1}, d^{n+1}) \xrightarrow{\alpha^{n+1}} (\sigma^{n+2}, d^{n+2})$, And the inductive hypothesis holds for the prefix of the execution with n transitions. Thus there exists either an execution $E_1' : (\sigma_1, \overline{d}^1) \longrightarrow \cdots \longrightarrow (\overline{\sigma}^{n+1}, \mathsf{error})$ or an execution $E_2' : (\sigma_1, \overline{d}^1) \longrightarrow \cdots \longrightarrow (\sigma^{n+1}, \overline{d}^{n+1})$ satisfying the claim. If the former holds, the execution E_1' of P_2 is a witness to the claim. Assume the latter in the following.

$\langle 5\rangle 2$. Consider the last transition $(\sigma^{n+1}, d^{n+1}) \xrightarrow{\alpha^{n+1}} (\sigma^{n+2}, d^{n+2})$ of E.

$\langle 5\rangle 3$. Assume that α^{n+1} is a transition via CHOOSE-FIRST for the statement $\alpha_1 \Box \alpha_2$. Since this transition does not modify the store, $\sigma^{n+1} = \sigma^{n+2}$. In this case, let $d^{n+1} = C[\alpha_1 \Box \alpha_2]$ and $d^{i+2} = C[\alpha_1]$, then letting $\overline{d}^{i+1} = C[\alpha_1 \oplus \alpha_2]$ makes the execution E_2' of P_2 from the inductive hypothesis a witness to the claim. The same reasoning holds for CHOOSE-SECOND.

$\langle 5\rangle 4$. If $\alpha^{n+1} = \alpha_1$, then existence of the transition $(\sigma^{n+1}, d^{n+1}) \xrightarrow{\alpha_1} (\sigma^{n+2}, d^{n+2})$, by the definition of $\oplus$, implies that there exists a transition $(\sigma^{n+1}, \overline{d}^{n+1}) \xrightarrow{\alpha_1 \oplus \alpha_2} (\sigma^{n+2}, \overline{d}^{n+2})$ or the transition $(\sigma^{n+1}, d^{n+1}) \xrightarrow{\alpha_1 \oplus \alpha_2} (\sigma^{n+1}, \mathsf{error})$. Extending the execution E_2' of P_2 with either transition yields a witness execution to the claim. The same reasoning holds if $\alpha^{n+1} = \alpha_2$.

$\langle 5\rangle 5$. In other cases than above, the execution E_2' of P_2 can be extended by the transition $(\sigma^{n+1}, \overline{d}^{n+1}) \xrightarrow{\alpha^{n+1}} (\sigma^{n+2}, \overline{d}^{n+2})$.

$\langle 5\rangle 6$. Q.E.D.

$\langle 3\rangle 1$. Q.E.D.

$$\frac{\begin{array}{c}\text{REDUCE-LOOP} \\ P, \mathcal{I} \vdash \alpha : m \qquad m \in \{\mathbb{R}, \mathbb{L}\} \qquad \phi \triangleright \tau \vdash \mathcal{I} \\ \vDash \phi \Rightarrow \tau[\mathit{Var}/\mathit{Var}'] \qquad \mathcal{I} \vdash \phi \triangleright \tau \circ \alpha \preceq \phi \triangleright \tau\end{array}}{P, \mathcal{I} \dashrightarrow P[\alpha^{\circlearrowleft} \mapsto \phi \triangleright \tau], \mathcal{I}}$$

CASE: REDUCE-LOOP

PROOF SKETCH:We take a two-phase approach. In the first phase we generate, from the original execution, a new execution that contains each instance of $\alpha^{\circlearrowleft}$ as a sequence of α transitions adjacent to each other. While collecting the actions comprising the entire evaluation of a loop together, we follow a similar reasoning to REDUCE-SEQUENTIAL. In the second phase, we replace the serial sequence of transitions with a single transition of the given specification $\phi \triangleright \tau$. Since $\phi \triangleright \tau$ simulates zero or more iterations of the loop, the resulting execution either goes wrong or ends at the same final state the original execution ends. Since $\phi \triangleright \tau$ simulates zero iteration of the loop, evaluation of the loop via LOOP-SKIP can also be replaced with a transition of $\phi \triangleright \tau$.

In the following proof, we will use the fact that the transitions by the operational semantics rules LOOP-SKIP and LOOP-ITER, which replace $\alpha^{\circlearrowleft}$ with either skip or $\alpha; \alpha^{\circlearrowleft}$, respectively, can commute to both directions since these transitions do not modify the store. LOOP-SKIP executes the loop as a single transition that keeps the store as is. Moreover, we can always represent the transition LOOP-ITER and the evaluation of α it unrolls with a single transition. Thus, we will skip handling of transitions by LOOP-SKIP and LOOP-ITER in the following.

$\langle 3 \rangle 1$. **First phase:** Consider a given execution $E : (\sigma_1, t_{main} : Main_1) \longrightarrow^* (\sigma_2, d)$ of P_1. In this phase, we sort the actions in the execution by moving mover actions, which at the end results in an execution in which transitions of every instance of $\alpha^{\circlearrowleft}$ appear adjacent.

$\langle 3 \rangle 2$. We first define a commit action for each loop instance $\alpha^{\circlearrowleft}$. If α is right-mover then the last transition of α is the commit action. If α is left-mover then the first transition of α is the commit action. If both cases hold, the commit action is arbitrarily chosen to be either the first or the last transition.

$\langle 3 \rangle 3$. We define $Sorted(E)$ as to be an execution containing the same set of actions in E and specifying a total ordering between actions in E in the following way: Let $E : (\sigma^1, d^1) \xrightarrow{\alpha^1} (\sigma^2, d^2) \xrightarrow{\alpha^2}\longrightarrow \cdots \xrightarrow{\alpha^n} (\sigma^n, d^n)$. Let $ToMerge(E)$ be the entire set of transitions of α executed on behalf of the evaluation of an instance of $\alpha^{\circlearrowleft}$. Let $commit(E, \alpha)$ be the commit action of the $\alpha^{\circlearrowleft}$ instance in which α appears. Define a total order $\ll_E$ between actions in E and a total order $\ll_{Sorted(E)}$ between actions in $Sorted(E)$.

$\alpha^i \ll_{Sorted(E)} \alpha^j$ iff

1) $\alpha^i \ll_E \alpha^j$ and $\alpha^i \notin ToMerge(E)$ and $\alpha^j \notin ToMerge(E)$,
2) $\alpha^i \ll_E \alpha^j$ and $iscommit(E, \alpha^i)$ and $\alpha^j \notin ToMerge(E)$,
3) $\alpha^i \ll_E \alpha^j$ and $\alpha^i \notin ToMerge(E)$ and $iscommit(E, \alpha^j)$,
4) $\alpha^i \ll_E \alpha^j$ and $iscommit(E, \alpha^i)$ and $iscommit(E, \alpha^j)$,
5) $\alpha^i \in ToMerge(E)$, $\alpha^j \notin ToMerge(E)$ and $commit(E, \alpha^i) \ll_{Sorted(E)} \alpha^j$,
6) $\alpha^i \notin ToMerge(E)$, $\alpha^j \in ToMerge(E)$ and $\alpha^i \ll_{Sorted(E)} commit(E, \alpha^j)$,
7) $\alpha^i, \alpha^j \in ToMerge(E)$ and $commit(E, \alpha^i) \ll_{Sorted(E)} commit(E, \alpha^j)$.

$\langle 3 \rangle 4$. Let $M(\cdot)$ be a measure function over executions such that $M(E)$ is the total number of inversions, i.e. $\alpha^i \ll_{Sorted(E)} \alpha^j$ but $\alpha^i \ll_E \alpha^j$. Each Each swap while *bubble-sorting* the execution E to get $Sorted(E)$ is done between a transition of a right-mover α or a left-mover α with another action α^i that involves an inversion. As we will show below that each swap either shortens the execution by causing it to go wrong earlier, or generates a new execution E' with $M(E') < M(E)$. Thus the fact that $M(E)$ for each execution E is finite implies that this phase terminates outputting a sorted execution (possibly shorter than original one). In the following, we will show the property of each swap that ensures this termination guarantee and the final form of the execution output from this phase.

PROVE: Let $E : (\sigma_1, t_{main} : Main_1) \longrightarrow^* (\sigma_2, d)$ be the current execution before a swap of α, which belongs to an instance of $\alpha^{\circlearrowleft}$, with another action α^i performed by a different thread. The swap generates either execution $E' : (\sigma_1, t_{main} : Main_1) \longrightarrow^* (\sigma_3, \mathsf{error})$ for some σ_3 (possibly shorter than E), or an execution $E' : (\sigma_1, t_{main} : Main_1) \longrightarrow^* (\sigma_2, d)$ such that $M(E') < M(E)$.

$\langle 4 \rangle 1$. Let current the execution be $E : (\sigma_1, t_{main} : Main_1) \longrightarrow^* (\sigma_2, d)$.

$\langle 4 \rangle 2$. There are two cases: Either $P, \mathcal{I} \vdash \alpha : \mathbb{R}$ and α moves to the right, or $P, \mathcal{I} \vdash \alpha : \mathbb{L}$ and α moves to the left.

CASE: $P_1, \mathcal{I} \vdash \alpha : \mathbb{R}$

$\langle 5 \rangle 1$. Assume that the execution is of the form: $(\sigma_1, t_{main} : Main_1) \longrightarrow \cdots \longrightarrow (\sigma^{i-1}) \xrightarrow{\alpha} (\sigma^i) \xrightarrow{\alpha^i} (\sigma^{i+1}) \longrightarrow \cdots \longrightarrow (\sigma_2, d)$. By the definition of right-movers, we can move the evaluation of α to the right of α^i to obtain another execution of P in the following way: The transition sequence $(\sigma^{i-1}) \xrightarrow{\alpha} (\sigma^i) \xrightarrow{\alpha^i} (\sigma^{i+1})$ can be replaced with either the transition $(\sigma^{i-1}) \xrightarrow{\alpha^i} (\sigma^{i-1}, \mathsf{error})$, or the transition sequence $(\sigma^{i-1}) \xrightarrow{\alpha^i} (\overline{\sigma}) \xrightarrow{\alpha} (\overline{\sigma}, \mathsf{error})$, or the transition sequence $(\sigma^{i-1}) \xrightarrow{\alpha^i} (\overline{\sigma}) \xrightarrow{\alpha} (\sigma^{i+1})$ for some $\overline{\sigma}$.

$\langle 5 \rangle 2$. In the first case, we obtain a shorter execution E' that goes wrong: $E' : (\sigma_1, t_{main} : Main_2) \longrightarrow \cdots \longrightarrow (\sigma^{i-1}) \xrightarrow{\alpha^i} (\sigma^{i-1}, \mathsf{error})$

$\langle 5 \rangle 3$. If the second case happens, we obtain an execution E' that goes wrong and $M(E') < M(E)$: $E' : (\sigma_1, t_{main} : Main_2) \longrightarrow \cdots \longrightarrow (\sigma^{i-1}) \xrightarrow{\alpha^i} (\overline{\sigma}) \xrightarrow{\alpha} (\overline{\sigma}, \mathsf{error})$.

$\langle 5 \rangle 4$. If the third case happens, we obtain an execution that goes to the same program state as the previous one: $(\sigma_1, t_{main} : Main_1) \longrightarrow \cdots \longrightarrow (\sigma^{i-1}) \xrightarrow{\alpha^i} (\overline{\sigma}) \xrightarrow{\alpha} (\sigma^{i+1}) \longrightarrow \cdots \longrightarrow (\sigma_2, d)$, by executing the rest of the execution from (σ^{i+1}) as in the previous execution. In this case $M(E') < M(E)$.

CASE: $P_1, \mathcal{I} \vdash \alpha : \mathbb{L}$

$\langle 5 \rangle 1$. Assume that the execution be of the form: $E : (\sigma_1, t_{main} : Main_1) \longrightarrow \cdots \xrightarrow{\alpha} \cdots \longrightarrow (\sigma^{i-1}) \xrightarrow{\alpha^i} (\sigma^i) \xrightarrow{\alpha} (\sigma^{i+1}) \longrightarrow \cdots \longrightarrow (\sigma_2, d)$. By the definition of left-movers, we can move the evaluation of α to the left of α^i to obtain another

execution of P in the following way: The transition sequence $(\sigma^{i-1}) \xrightarrow{\alpha^i} (\sigma^i) \xrightarrow{\alpha} (\sigma^{i+1})$ can be replaced with either the transition $(\sigma^{i-1}) \xrightarrow{\alpha} (\sigma^{i-1}, \mathsf{error})$, or the transition sequence $(\sigma^{i-1}) \xrightarrow{\alpha} (\overline{\sigma}) \xrightarrow{\alpha^i} (\overline{\sigma}, \mathsf{error})$ for some $\overline{\sigma}$ or the transition sequence $(\sigma^{i-1}) \xrightarrow{\alpha} (\overline{\sigma}) \xrightarrow{\alpha^i} (\sigma^{i+1})$.

⟨5⟩2. In the first case, we obtain an execution that goes wrong before executing α^i: $E : (\sigma_1, t_{main} : Main_1) \longrightarrow \cdots \xrightarrow{\alpha} \cdots \longrightarrow (\sigma^{i-1}) \xrightarrow{\alpha} (\sigma^i, \mathsf{error})$. Thus E' is shorter than E.

⟨5⟩3. In the second case, we obtain an execution that goes wrong: $(\sigma_1, t_{main} : Main_1) \longrightarrow \cdots \xrightarrow{\alpha} \cdots \longrightarrow (\sigma^{i-1}) \xrightarrow{\alpha} (\overline{\sigma}) \xrightarrow{\alpha^i} (\overline{\sigma}, \mathsf{error})$. In this case $M(E') < M(E)$.

⟨5⟩4. If the third case happens, we obtain an execution that goes to the same program state as the previous one: $(\sigma_1, t_{main} : Main_1) \longrightarrow \cdots \xrightarrow{\alpha} \cdots \longrightarrow (\sigma^{i-1}) \xrightarrow{\alpha} (\overline{\sigma}) \xrightarrow{\alpha^i} (\sigma^{i+1}) \longrightarrow \cdots \longrightarrow (\sigma_2, d)$, by executing the rest of the execution from (σ^{i+1}) as in the previous execution. In this case $M(E') < M(E)$.

⟨4⟩3. Q.E.D.

⟨3⟩5. We perform the above iteration until we obtain either an execution $E' : (\sigma_1, t_{main} : Main_1) \longrightarrow^* (\sigma_3, \mathsf{error})$ for some σ_3, or an execution $E' : (\sigma_1, t_{main} : Main_1) \longrightarrow^* (\sigma_2, d)$. In both cases $M(E') = 0$.

⟨3⟩6. **Second phase:** In this phase, we replace the serial sequence of transitions of each loop instance with the specification of the loop $\phi \triangleright \tau$. After each replacement, the resulting execution either goes wrong at a store, or ends at the same program state (σ_2, d) as the original execution.

PROVE: For each execution $(\sigma_1, t_{main} : Main_1) \longrightarrow^* (\sigma_2, d)$, there exists either an execution $(\sigma_1, t_{main} : Main_1) \longrightarrow \cdots \longrightarrow (\sigma_3, \overline{d})$ or an execution $(\sigma_1, t_{main} : Main_1) \longrightarrow \cdots \longrightarrow (\overline{\sigma}, \mathsf{error})$ for some $\overline{\sigma}$ of P_2, where $\overline{d}$ is d except for all occurrences of $\alpha^{\circlearrowleft}$ is replaced with $\phi \triangleright \tau$.

⟨4⟩1. Proof by induction on the number n of transitions in the prefix.

CASE: Base case: For $n = 0$, the same execution is a witness to the claim.

CASE: Inductive case: Suppose that the claim holds for all prefixes of the execution of length less than or equal to n. We will prove that the claim holds for all prefixes with length $n + 1$.

⟨5⟩1. Assume that the execution is of the form $E : (\sigma_1, t_{main} : Main_1) \longrightarrow \cdots \longrightarrow (\sigma^{n+1}, d^{n+1}) \xrightarrow{\alpha^{n+1}} (\sigma^{n+2}, d^{n+2})$ of length $n+1$. By inductive hypothesis, we know that there exists either an execution $E_1' : (\sigma_1, t_{main} : Main_2) \longrightarrow \cdots \longrightarrow (\overline{\sigma}, \mathsf{error})$, for some $\overline{\sigma}$, or an execution $E_2' : (\sigma_1, t_{main} : Main_2) \longrightarrow \cdots \longrightarrow (\sigma^{n+1}, \overline{d}^{n+1})$ of P_2. If the former holds, the execution E_1' of P_2 is also a witness of the claim for the given execution of P_1. In the following assume the latter.

⟨5⟩2. Consider the last transition of E: $(\sigma^{n+1}, d^{n+1}) \xrightarrow{\alpha^{n+1}} (\sigma^{n+2}, d^{n+2})$. If α^{n+1} is a transition that evaluates an instance of the statement $\alpha^{\circlearrowleft}$ via LOOP-SKIP, we can extend the execution E_2' of P_2 with a transition of $\phi \triangleright \tau$. By the premise of the rule, $\phi \triangleright \tau$ either causes the execution to go wrong or otherwise can take an identity transition, as taken by LOOP-SKIP. Both cases yield a witness execution.

⟨5⟩3. If $\alpha^n \neq \alpha$ and $\alpha^{n+1} = \alpha$, then we derive a witness execution by changing the last state of E_2' as follows: $(\sigma_1, t_{main} : Main_2) \longrightarrow \cdots (\sigma^{n+1}, \overline{d}^{n+1}) \xrightarrow{\phi \triangleright \tau} (\sigma^{n+2}, \overline{d}^{n+2})$. This is valid since, by the premise of the rule, a transition sequence of α is simulated by $\phi \triangleright \tau$: If α goes wrong from σ^{n+1} then $\phi \triangleright \tau$ also goes wrong from σ^{n+1}, otherwise $(\sigma^{n+1}, \overline{d}^{n+1}) \xrightarrow{\phi \triangleright \tau} (\sigma^{n+2}, \overline{d}^{n+2})$ can be derived.

⟨5⟩4. Now assume that $\alpha^{n+1} = \alpha$. If $\alpha^n = \phi \triangleright \tau$, then we derive a witness execution by changing the last state of E_2' as follows: $(\sigma_1, t_{main} : Main_2) \longrightarrow \cdots \longrightarrow (\sigma^{n+2}, \overline{d}^{n+1})$. This is valid since, by the premise of the rule, a transition sequence of $\phi \triangleright \tau$ and α is simulated by $\phi \triangleright \tau$: If α goes wrong from σ^{n+1}, then $\phi \triangleright \tau$ also goes wrong from σ^n, otherwise $(\sigma^n, \overline{d}^n) \xrightarrow{\phi \triangleright \tau} (\sigma^{n+2}, \overline{d}^{n+1})$ can be derived.

⟨5⟩5. If α^{n+1} is an evaluation of LOOP-ITER, then E_2' is a witness execution, since LOOP-ITER takes identity transitions on the store.

⟨5⟩6. In any other cases, we extend E_2' with the last transition of E.

⟨4⟩2. Q.E.D.

⟨3⟩7. The above claim implies that we can replace the output of the first phase with a witness execution of P_2 to the claim, that either goes to the same store as P_1 or goes wrong, both from the same store.

⟨3⟩8. Q.E.D.

$$\text{EXPAND-PARALLEL}\quad \frac{\alpha_3 = \alpha_1[left(tid)/tid] \qquad \alpha_4 = \alpha_2[right(tid)/tid]}{P, \mathcal{I} \dashrightarrow P[\alpha_1 \| \alpha_2 \mapsto (\alpha_3; \alpha_4) \square (\alpha_4; \alpha_3)], \mathcal{I}}$$

CASE: EXPAND-PARALLEL

PROOF SKETCH: Show that in each execution transitions of α_1 and α_2 can be replaced by transitions of α_3 and α_4, respectively, since the transitions via FORK labels α_1 and α_2 to get α_3 and α_4 before evaluating. The new statement $(\alpha_3; \alpha_4) \square (\alpha_4; \alpha_3)$ allows this to be done considering evaluation of $\alpha_1 \| \alpha_2$ in any order.

⟨4⟩1. If $\alpha_1 \| \alpha_2$ is not evaluated at all, the same execution can be used as a witness to the claim (where $(\alpha_3; \alpha_4) \square (\alpha_4; \alpha_3)$ is not evaluated at all).

⟨4⟩2. Let $tl = left(t)$ and $tr = right(t)$. Assume an execution $(\sigma_1, t_{main} : Main_1) \longrightarrow \cdots \longrightarrow (\sigma_3, u : C[t : (\alpha_1 \| \alpha_2)]) \longrightarrow (\sigma_3, u : C[tl : \alpha_1 \| tr : \alpha_2]) \longrightarrow \cdots \longrightarrow (\sigma_2, d)$ where $d \in \{\mathsf{skip}, \mathsf{error}\}$.

⟨4⟩3. We can derive, for each such execution given in ⟨4⟩2, a witness execution $(\sigma_1, Main_2) \longrightarrow \cdots \longrightarrow (\sigma_3, u : C[t : ((\alpha_3; \alpha_4) \square (\alpha_4; \alpha_3))]) \longrightarrow \cdots \longrightarrow (\sigma_2, d)$ by taking the following three actions:

⟨5⟩1. Remove the transitions (if any) by the operational semantics rules FORK, JOIN-FIRST and JOIN-SECOND (which replaces $\mathsf{skip}\|\alpha_2$ with α_2 or $\alpha_1\|\mathsf{skip}$ with α_1), since these transitions can do not modify the store.

⟨5⟩2. Replace transitions performed by α_1 and α_2 with α_3 and α_4, respectively. This preserves the same pre- and post-stores of the regarding transitions since evaluating $tl : \alpha_1$ by ATOMIC and ERROR is equivalent to evaluating $t : \alpha_1[left(tid)/tid]$ by these rules. Similarly, evaluating $tr : \alpha_2$ by ATOMIC and ERROR is equivalent to evaluating $t : \alpha_2[right(tid)/tid]$ by those rules.

⟨5⟩3. Insert into the execution transitions by LABEL, SEQUENTIAL, CHOOSE-FIRST and CHOOSE-SECOND to evaluate the sequential and choice compositions. These transitions do not modify the store and only required to conform with the operational semantics.

⟨3⟩1. Q.E.D.

REDUCE-PARALLEL-I*
$$\frac{P,\mathcal{I}\vdash \alpha_1 : \mathbb{L} \quad \text{or} \quad P,\mathcal{I}\vdash \alpha_2 : \mathbb{R} \qquad \alpha_3 = \alpha_1[left(tid)/tid] \quad \alpha_4 = \alpha_2[right(tid)/tid]}{P,\mathcal{I} \dashrightarrow P[\alpha_1\|\alpha_2 \mapsto \alpha_3;\alpha_4],\mathcal{I}}$$

REDUCE-PARALLEL-II*
$$\frac{P,\mathcal{I}\vdash \alpha_1 : \mathbb{R} \quad \text{or} \quad P,\mathcal{I}\vdash \alpha_2 : \mathbb{L} \qquad \alpha_3 = \alpha_1[left(tid)/tid] \quad \alpha_4 = \alpha_2[right(tid)/tid]}{P,\mathcal{I} \dashrightarrow P[\alpha_1\|\alpha_2 \mapsto \alpha_4;\alpha_3],\mathcal{I}}$$

CASE: REDUCE-PARALLEL-I/II

PROOF SKETCH:After applying the rule EXPAND-PARALLEL, reason about that a witness execution in which these actions will be executed in the given ordering of α_3 and α_4 can be derived for each execution in which these actions are executed in the missing orderings. For this, we can do a similar reasoning to the one done while proving REDUCE-SEQUENTIAL.

⟨4⟩1. By the proof of the rule EXPAND-PARALELL, we know that for each execution $(\sigma_1, t_{main} : Main_1) \longrightarrow \cdots \longrightarrow (\sigma_3, u : C[t : (\alpha_1\|\alpha_2)]) \longrightarrow (\sigma_3, u : C[tl : \alpha_1\|tr : \alpha_2]) \longrightarrow \cdots \longrightarrow (\sigma_2, d)$ where $d \in \{\mathsf{skip}, \mathsf{error}\}$, there is an execution $(\sigma_1, Main_1) \longrightarrow \cdots \longrightarrow (\sigma_3, u : C[t : ((\alpha_3;\alpha_4)\Box(\alpha_4;\alpha_3))]) \longrightarrow \cdots \longrightarrow (\sigma_2, d)$.

⟨4⟩2. Consider the rule REDUCE-PARALLEL-I. The transitions via FORK labels α_1 and α_2 to get α_3 and α_4 before evaluating, respectively. $P,\mathcal{I} \vdash \alpha_3 : \mathbb{L}$ or $P,\mathcal{I} \vdash \alpha_4 : \mathbb{R}$. The executions of P_1 where α_3 is evaluated before α_4 are witnesses for P_2, for those executions transitions of $\alpha_3;\alpha_4$ simulates the transitions of $(\alpha_3;\alpha_4)\Box(\alpha_4;\alpha_3)$ after removing the transitions via CHOOSE-FIRST. Assume that in the derived execution α_4 is evaluated before α_3. By following a similar reasoning done while proving REDUCE-SEQUENTIAL, we can move α_4 to the right (if $P,\mathcal{I} \vdash \alpha_4 : \mathbb{R}$ holds) or α_3 to the left (if $P,\mathcal{I} \vdash \alpha_3 : \mathbb{L}$ holds) until we obtain an execution in which α_3 is executed before α_4. This final execution either goes wrong, or ends in the same program state (σ_2, d) of the original execution, thus it is a witness to both Condition *1* and *2*. of the lemma.

⟨4⟩3. Now consider the rule REDUCE-PARALLEL-II. The transitions via FORK labels α_1 and α_2 to get α_3 and α_4 before evaluating, respectively. $P,\mathcal{I} \vdash \alpha_3 : \mathbb{R}$ or $P,\mathcal{I} \vdash \alpha_4 : \mathbb{L}$. The executions of P_1 where α_4 is evaluated before α_3 are witnesses for P_2, for those executions transitions of $\alpha_4;\alpha_3$ simulates the transitions of $(\alpha_3;\alpha_4)\Box(\alpha_4;\alpha_3)$ after removing the transitions via CHOOSE-SECOND. Assume that in the derived execution α_3 is evaluated before α_4. By following a similar reasoning done while proving REDUCE-SEQUENTIAL, we can move α_3 to the right (if $P,\mathcal{I} \vdash \alpha_3 : \mathbb{R}$ holds) or α_4 to the left (if $P,\mathcal{I} \vdash \alpha_4 : \mathbb{L}$ holds) until we obtain an execution in which α_3 is executed before α_4. This final execution either goes wrong, or ends in the same program state (σ_2, d) of the original execution, thus it is a witness to both Condition *1* and *2*. of the lemma.

⟨3⟩1. Q.E.D.

INLINE-CALL
$$\frac{body(\rho) = c}{P,\mathcal{I} \dashrightarrow P[\rho() \mapsto c],\mathcal{I}}$$

CASE: INLINE-CALL

⟨4⟩1. Given an execution of P_1, removing the transition by PROC-CALL that replaces $\rho()$ with $body(\rho)$, yields a witness execution of P_2 to the claims in the lemma. This is safe since transitions by PROC-CALL does not modify the store. If no such transitions exists then it means that the procedure is not evaluated, so the same executions is a witness of P_2 to the claims of the lemma.

⟨3⟩1. Q.E.D.

REDUCE-PROCEDURE
$$\frac{\begin{array}{c}\mathcal{M} = \{\rho_1, ..., \rho_n\} \text{ is closed under call} \qquad \text{There exists a function } Annot \text{ s.t. } Annot(\rho_i) = (\phi_i \triangleright \tau_i, m_i) \\ \forall \rho_i \in \mathcal{M} : \quad P,\mathcal{I}, Annot \vdash \{\phi_i \triangleright \tau_i\}\rho_i \qquad P,\mathcal{I}, Annot \vdash body(\rho) : m_i \qquad \phi_i \triangleright \tau_i \vdash \mathcal{I} \\ \text{Every execution of } \rho_i \text{ from } \phi_i \text{ can be extended to a terminating or blocking execution.}\end{array}}{P,\mathcal{I} \dashrightarrow P[body(\rho_1) \mapsto \phi_1 \triangleright \tau_1, \ldots, body(\rho_n) \mapsto \phi_n \triangleright \tau_n],\mathcal{I}}$$

CASE: REDUCE-PROCEDURE

ASSUME: For the proof of this rule, we will make a termination assumption about procedures in $\mathcal{M}$: Let $Annot(\rho_i) = \phi_i \triangleright \tau_i$. Any execution of a procedure $\rho_i \in \mathcal{M}$ from any store satisfying ϕ_i can be extended to either a terminating execution or a blocking execution[5].

⟨3⟩1. For each execution let $\hookrightarrow$ be a partial order on the procedure calls that happen in the execution such that $\rho_1() \hookrightarrow \rho_2()$ if evaluation of $\rho_1()$ calls ρ_2 in the execution. A procedure call $\rho_x()$ is called a *root* call if there is no $\rho_y()$ such that $\rho_y \in \mathcal{M}$ and $(\rho_y() \hookrightarrow \rho_x())$.

PROOF SKETCH:Given an execution $(\sigma_1, t_{main} : Main_1) \longrightarrow^* (\sigma_2, d)$ and $\hookrightarrow$ of the execution as defined above, we will generate a witness execution in 3 phases. In each phase we will iteratively generate executions either in the form of $(\sigma_1, t_{main} : Main_1) \longrightarrow^* (\sigma_2, d)$ or in the form of $(\sigma_1, t_{main} : Main_1) \longrightarrow^* (\sigma_3, \mathsf{error})$ for some σ_3. In the first phase, similar to the first phase of the proof of REDUCE-SEQUENTIAL we will construct an execution in which evaluation of each root procedure call appears sequential. In the second

[5] A terminating execution ends at a state (σ, d) for some σ, where $d \in \{\mathsf{skip}, \mathsf{error}\}$. A blocking execution ends at a state (σ, d) for some σ, where $d \notin \{\mathsf{skip}, \mathsf{error}\}$, from which no transition can be derived by the operational semantics rules.

phase, we will obtain an execution in which evaluation of every root procedure call is completed either with skip or error. In the third phase, we will replace the evaluations of root calls with transitions of their specification in $Annot$.

$\langle 4 \rangle 1$. **First phase:** Consider a given execution $E : (\sigma_1, t_{main} : Main_1) \longrightarrow^* (\sigma_2, d)$ of P_1. In this phase, we move the actions of root calls in the execution, resulting in an execution in which all transitions of every root call $\rho_i()$ appear adjacent to each other.

$\langle 4 \rangle 2$. We now define a commit action for each root call $\rho_i()$. If the root call is in its right-mover phase of computation, i.e. if all its actions in the execution are right-movers, then the commit action is the last action of $\rho_i()$ in the execution. If the root call is in its left-mover phase of computation, i.e. if it has executed at least one action that is not a right-mover, then the commit action is the first non-right-mover of $\rho_i()$ in the execution. Let $Sorted(E)$ be a form of the given execution such that transitions of all root calls in E appear in $Sorted(E)$ adjacent to each other with the order of their commit actions in E, and order of other actions relative to each other and the commit actions remain the same as in E.

$\langle 4 \rangle 3$. We define $Sorted(E)$ as to be an execution containing the same set of actions in E and specifying a total ordering between actions in E in the following way: Let $E : (\sigma^1, d^1) \xrightarrow{\alpha^1} (\sigma^2, d^2) \xrightarrow{\alpha^2}\longrightarrow \cdots \xrightarrow{\alpha^n} (\sigma^n, d^n)$. Let $ToMerge(E)$ be the entire set of transitions of executed on behalf of all root calls in E. Let $commit(E, \alpha)$ be the commit action of the root call in which α appears. Define a total order $\ll_E$ between actions in E and a total order $\ll_{Sorted(E)}$ between actions in $Sorted(E)$.
$\alpha^i \ll_{Sorted(E)} \alpha^j$ iff
1) $\alpha^i \ll_E \alpha^j$ and $\alpha^i \notin ToMerge(E)$ and $\alpha^j \notin ToMerge(E)$,
2) $\alpha^i \ll_E \alpha^j$ and $iscommit(E, \alpha^i)$ and $\alpha^j \notin ToMerge(E)$,
3) $\alpha^i \ll_E \alpha^j$ and $\alpha^i \notin ToMerge(E)$ and $iscommit(E, \alpha^j)$,
4) $\alpha^i \ll_E \alpha^j$ and $iscommit(E, \alpha^i)$ and $iscommit(E, \alpha^j)$,
5) $\alpha^i \in ToMerge(E)$, $\alpha^j \notin ToMerge(E)$ and $commit(E, \alpha^i) \ll_{Sorted(E)} \alpha^j$,
6) $\alpha^i \notin ToMerge(E)$, $\alpha^j \in ToMerge(E)$ and $\alpha^i \ll_{Sorted(E)} commit(E, \alpha^j)$,
7) $\alpha^i, \alpha^j \in ToMerge(E)$ and $commit(E, \alpha^i) \ll_{Sorted(E)} commit(E, \alpha^j)$.

$\langle 4 \rangle 4$. Let $M(\cdot)$ be a measure function over executions such that $M(E)$ is the total number of inversions, i.e. $\alpha^i \ll_{Sorted(E)} \alpha^j$ but $\alpha^i \ll_E \alpha^j$. Each Each swap while *bubble-sorting* the execution E to get $Sorted(E)$ is done between a transition of a right-mover or a left-mover action of a root call with another action α^i that involves an inversion. As we will show below that each swap either shortens the execution by causing it to go wrong earlier, or generates a new execution E' with $M(E') < M(E)$. Thus the fact that $M(E)$ for each execution E is finite implies that this phase terminates outputting a sorted execution (possibly shorter than original one). In the following, we will show the property of each swap that ensures this termination guarantee and the final form of the execution output from this phase.

PROVE: Let $E : (\sigma_1, t_{main} : Main_1) \longrightarrow^* (\sigma_2, d)$ be the current execution before a swap of an action α that belongs to a root call's evaluation with another action α^i performed by a different thread. The swap generates either execution $E' : (\sigma_1, t_{main} : Main_1) \longrightarrow^* (\sigma_3, \mathsf{error})$ for some σ_3 (possibly shorter than E), or an execution $E' : (\sigma_1, t_{main} : Main_1) \longrightarrow^* (\sigma_2, d)$ such that $M(E') < M(E)$.

$\langle 5 \rangle 1$. Let current the execution be $E : (\sigma_1, t_{main} : Main_1) \longrightarrow^* (\sigma_2, d)$.

$\langle 5 \rangle 2$. There are two cases: Either $P, \mathcal{I} \vdash \alpha : \mathbb{R}$ and α moves to the right, or $P, \mathcal{I} \vdash \alpha : \mathbb{L}$ and α moves to the left.

CASE: $P_1, \mathcal{I} \vdash \alpha : \mathbb{R}$

$\langle 6 \rangle 1$. Assume that the execution is of the form: $(\sigma_1, t_{main} : Main_1) \longrightarrow \cdots \longrightarrow (\sigma^{i-1}) \xrightarrow{\alpha} (\sigma^i) \xrightarrow{\alpha^i} (\sigma^{i+1}) \longrightarrow \cdots \longrightarrow (\sigma_2, d)$. By the definition of right-movers, we can move the evaluation of α to the right of α^i to obtain another execution of P in the following way: The transition sequence $(\sigma^{i-1}) \xrightarrow{\alpha} (\sigma^i) \xrightarrow{\alpha^i} (\sigma^{i+1})$ can be replaced with either the transition $(\sigma^{i-1}) \xrightarrow{\alpha^i} (\sigma^{i-1}, \mathsf{error})$, or the transition sequence $(\sigma^{i-1}) \xrightarrow{\alpha^i} (\overline{\sigma}) \xrightarrow{\alpha} (\overline{\sigma}, \mathsf{error})$, or the transition sequence $(\sigma^{i-1}) \xrightarrow{\alpha^i} (\overline{\sigma}) \xrightarrow{\alpha} (\sigma^{i+1})$ for some $\overline{\sigma}$.

$\langle 6 \rangle 2$. In the first case, we obtain a shorter execution E' that goes wrong: $E' : (\sigma_1, t_{main} : Main_2) \longrightarrow \cdots \longrightarrow (\sigma^{i-1}) \xrightarrow{\alpha^i} (\sigma^{i-1}, \mathsf{error})$

$\langle 6 \rangle 3$. If the second case happens, we obtain an execution E' that goes wrong and $M(E') < M(E)$: $E' : (\sigma_1, t_{main} : Main_2) \longrightarrow \cdots \longrightarrow (\sigma^{i-1}) \xrightarrow{\alpha^i} (\overline{\sigma}) \xrightarrow{\alpha} (\overline{\sigma}, \mathsf{error})$.

$\langle 6 \rangle 4$. If the third case happens, we obtain an execution that goes to the same program state as the previous one: $(\sigma_1, t_{main} : Main_1) \longrightarrow \cdots \longrightarrow (\sigma^{i-1}) \xrightarrow{\alpha^i} (\overline{\sigma}) \xrightarrow{\alpha} (\sigma^{i+1}) \longrightarrow \cdots \longrightarrow (\sigma_2, d)$, by executing the rest of the execution from (σ^{i+1}) as in the previous execution. In this case $M(E') < M(E)$.

CASE: $P_1, \mathcal{I} \vdash \alpha : \mathbb{L}$

$\langle 6 \rangle 1$. Assume that the execution be of the form: $E : (\sigma_1, t_{main} : Main_1) \longrightarrow \cdots \xrightarrow{\alpha} \cdots \longrightarrow (\sigma^{i-1}) \xrightarrow{\alpha^i} (\sigma^i) \xrightarrow{\alpha} (\sigma^{i+1}) \longrightarrow \cdots \longrightarrow (\sigma_2, d)$. By the definition of left-movers, we can move the evaluation of α to the left of α^i to obtain another execution of P in the following way: The transition sequence $(\sigma^{i-1}) \xrightarrow{\alpha^i} (\sigma^i) \xrightarrow{\alpha} (\sigma^{i+1})$ can be replaced with either the transition $(\sigma^{i-1}) \xrightarrow{\alpha} (\sigma^{i-1}, \mathsf{error})$, or the transition sequence $(\sigma^{i-1}) \xrightarrow{\alpha} (\overline{\sigma}) \xrightarrow{\alpha^i} (\overline{\sigma}, \mathsf{error})$ for some $\overline{\sigma}$ or the transition sequence $(\sigma^{i-1}) \xrightarrow{\alpha} (\overline{\sigma}) \xrightarrow{\alpha^i} (\sigma^{i+1})$.

$\langle 6 \rangle 2$. In the first case, we obtain an execution that goes wrong before executing α^i: $E : (\sigma_1, t_{main} : Main_1) \longrightarrow \cdots \xrightarrow{\alpha} \cdots \longrightarrow (\sigma^{i-1}) \xrightarrow{\alpha} (\sigma^i, \mathsf{error})$. Thus E' is shorter than E.

⟨6⟩3. In the second case, we obtain an execution that goes wrong: $(\sigma_1, t_{main} : Main_1) \longrightarrow \cdots \xrightarrow{\alpha} \cdots \longrightarrow (\sigma^{i-1}) \xrightarrow{\alpha} (\overline{\sigma}) \xrightarrow{\alpha^i} (\overline{\sigma}, \mathsf{error})$. In this case $M(E') < M(E)$.

⟨6⟩4. If the third case happens, we obtain an execution that goes to the same program state as the previous one: $(\sigma_1, t_{main} : Main_1) \longrightarrow \cdots \xrightarrow{\alpha} \cdots \longrightarrow (\sigma^{i-1}) \xrightarrow{\alpha} (\overline{\sigma}) \xrightarrow{\alpha^i} (\sigma^{i+1}) \longrightarrow \cdots \longrightarrow (\sigma_2, d)$, by executing the rest of the execution from (σ^{i+1}) as in the previous execution. In this case $M(E') < M(E)$.

⟨5⟩3. Q.E.D.

⟨4⟩5. We perform the above iteration until we obtain either an execution $E' : (\sigma_1, t_{main} : Main_1) \longrightarrow^* (\sigma_3, \mathsf{error})$ for some σ_3, or an execution $E' : (\sigma_1, t_{main} : Main_1) \longrightarrow^* (\sigma_2, d)$. In both cases $M(E') = 0$.

⟨4⟩6. **Second phase:** We need this phase only if the given execution goes wrong, i.e. in the form of $(\sigma_1, t_{main} : Main_1) \longrightarrow^* (\sigma_3, \mathsf{error})$ for some σ_3. If the current execution terminates normally, i.e. goes to $(\sigma_2, \mathsf{skip})$, we skip this phase. In this phase, we apply an iterative process that follows a similar reasoning done while proving the lemma for REDUCE-SEQUENTIAL. Our aim is to have every root call with a terminating evaluation, either with skip or error, in the final execution of this phase. This is required because the third phase replaces sequential evaluations of root calls with the transitions of their specifications, each of which specify only terminating behaviors of the regarding procedure.

⟨4⟩7. Let $\Rsh$ denote a total order between the root calls in the output execution of the first phase, such that $\rho_i \Rsh \rho_j$ if the (serial sequence of) actions of ρ_i appear before the actions of ρ_j in the execution.

⟨4⟩8. We first eliminate from the execution all the root calls in their right-mover phase, except for one last root-call that leads the execution to error, as described below.

⟨5⟩1. At each step, we take the leftmost root call, say $\rho_i()$ in its right-mover phase, and move one action, say α, that is executed on behalf of $\rho_i()$ to the right. This ends when an execution that goes wrong earlier without executing α. Each move either eliminates α from the execution, or causes the execution to go wrong early, both of which reduces the length of the execution. If the last action of the input execution, which goes wrong, does not belong to a root call, then the final execution contains no root calls in right-mover phase, otherwise the final execution contains only a single root call in its right-mover phase, which leads the execution to error. The former happens because for each action α of a root call in its right-mover phase, we can always generate an execution that goes wrong without executing α. See the description of the first phase above for a formal reasoning about the possible outcomes of moving a right-mover to the right in the execution.

⟨4⟩9. We now have an execution that is of the following form: All the root calls except for the last one in the $\Rsh$ order are in their left-mover phase; the last root call is either in its left-mover or right-mover phase. If the latter holds, the transitions of the last root call are the last transitions of the execution that lead the execution to error. Note that, we do not need to complete this root call, since its evaluation already terminates with error; we only complete other root calls, which are in their left-mover phases.

⟨5⟩1. At each iteration of this phase, we take the leftmost root call $\rho_i()$ (such that $\nexists \rho_j().\ \rho_j \in \mathcal{M} \wedge \rho_j() \Rsh \rho_i()$). Let $(\sigma, C[\rho()])$ be the program state in the current execution from which $\rho_i()$ starts executing, and $\phi_i \triangleright \tau_i$ be its specification in $Annot$. Since we always complete calls from the left to the right, we know that we have completed all calls $\rho_j()$ such that $\rho_j() \Rsh \rho_i()$. We do the followings to *complete* the call $\rho_i()$:

⟨5⟩2. We check whether the current state $(\sigma, C[\rho()])$ satisfies the gate of the specification $\phi_i \triangleright \tau_i$.

⟨6⟩1. If $\sigma \vDash \neg\phi_i$ then the prefix of the current execution up to the state $(\sigma, C[\rho()])$ is a witness execution of P_2 to the claim; replacing the call $\rho_i()$ with $\phi_i \triangleright \tau_i$ in P_2 will ensure that there exists an execution $(\sigma_1, t_{main} : Main_1) \longrightarrow^* (\sigma_3, \mathsf{error})$ for some σ_3 that goes wrong when evaluating $\phi_i \triangleright \tau_i$ from s state $(\sigma_3, t : C[\phi_i \triangleright \tau_i])$.

⟨6⟩2. If $\sigma \vDash \phi_i$, then we take the next action, say α, of the call $\rho_i()$ that has not been executed in the current execution, and extend the execution of $\rho_i()$ with α. Note that α is a left-mover, since we are only completing root calls in their left-mover phases. We do this by first generating an execution that contains an evaluation of α and then moving α to the left until it becomes adjacent to the last action of $\rho_i()$. α can be included in the execution, by definition of left-movers, by replacing the last transition $(\sigma^n, d^n) \xrightarrow{\alpha^n} (\sigma, \mathsf{error})$ of an execution with either $(\sigma^n, d^n) \xrightarrow{\alpha} (\sigma, \mathsf{error})$ or $(\sigma^n, d^n) \xrightarrow{\alpha} (\overline{\sigma}) \xrightarrow{\alpha^i} (\overline{\sigma}, \mathsf{error})$ for some σ. Including α in this way and moving it to the left, we either generate a shorter execution that goes wrong, or an execution of the same length, both with an α closer to the other transitions of $\rho_i()$. By our termination assumption, this process will terminate after extending the execution of $\rho_i()$ to an execution that ends with skip or error, or to a blocking execution. In fact, the last case cannot happen, since we can always keep adding left-movers to the end of $\rho_i()$. Thus, this process ends by completing the evaluation of $\rho_i()$ to a transition sequence that represents an execution of ρ_i that ends with skip or error.

⟨4⟩10. The output of this phase is an execution either in the form of $(\sigma_1, t_{main} : Main_1) \longrightarrow^* (\sigma_2, d)$ or in the form of $(\sigma_1, t_{main} : Main_1) \longrightarrow^* (\sigma_3, \mathsf{error})$ for some σ_3, and contains a *completed* evaluation of each root procedure call as a serial sequence of transitions by the same thread. Note that the serial evaluation of all the root procedures represents an execution of ρ_i that ends with skip or error.

⟨4⟩11. **Third phase:** In this phase, we replace the serial transition sequence of each root procedure call $\rho_i()$ with a transition of $\phi_i \triangleright \tau_i$. We do this by following the $\Rsh$ ordering of the root calls. Since each such transition sequence represents a terminating execution of ρ_i with skip or error, by the premise $P, \mathcal{I}, Annot \vdash \{\phi_i \triangleright \tau_i\}\rho_i$ of the rule, the following holds:

PROVE: Take an execution $(\sigma_1, t_{main} : Main_1) \longrightarrow^* (\sigma_2, d)$. For each prefix $(\sigma_1, t_{main} : Main_1) \longrightarrow \cdots \longrightarrow (\sigma_3, d_3)$ of the execution that contains completed root calls, there exists either an execution $(\sigma_1, t_{main} : Main_2) \longrightarrow \cdots \longrightarrow (\sigma_3, \overline{d}_3)$ or an execution $(\sigma_1, t_{main} : Main_2) \longrightarrow \cdots \longrightarrow (\overline{\sigma}, \mathsf{error})$ for some $\overline{\sigma}$ of P_2, where $\overline{d}_3$ is d_3 except for all occurrences of a root call $\rho_i()$ is replaced with its specification $\phi_i \triangleright \tau_i$.

⟨5⟩1. Proof by induction on the number n of the root calls.

CASE: Base case: For $n = 0$, the prefix of the execution before the first root call is a witness to the claim.

CASE: Inductive case: Suppose that the claim holds for all prefixes of the execution of containing less than or equal to n root calls. We will prove that the claim holds for all prefixes with length $n+1$.

⟨6⟩1. Take an execution $E : (\sigma_1, t_{main} : Main_1) \longrightarrow \cdots \longrightarrow (\sigma_3, d_3)$ and assume that every root call in E terminates either with skip or error. We will now reason about an extension $\overline{E}$ of the execution with entire sequence of transitions of a root call $\rho_i()$: $\overline{E} : (\sigma_1, t_{main} : Main_1) \longrightarrow \cdots \longrightarrow (\sigma_3, d_3) \xrightarrow{\alpha^1}\xrightarrow{\alpha^2} \cdots \xrightarrow{\alpha^n} (\sigma_4, d_4)$.

⟨6⟩2. By inductive hypothesis, we know that there exists either an execution $E_1' : (\sigma_1, t_{main} : Main_2) \longrightarrow \cdots \longrightarrow (\overline{\sigma}, \mathsf{error})$, for some $\overline{\sigma}$, or an execution $E_2' : (\sigma_1, t_{main} : Main_2) \longrightarrow \cdots \longrightarrow (\sigma_3, \overline{d}_3)$ of P_2.

⟨6⟩3. If the former holds, then E_1' is a witness for all extensions of E including $\overline{E}$.

⟨6⟩4. If the latter holds, then we extend the execution E_2' of P_2 with a transition of $\phi_i \triangleright \tau_i$, which yields, by the premise of the rule, either an execution $(\sigma_1, t_{main} : Main_2) \longrightarrow \cdots \longrightarrow (\sigma_3, \overline{d}_3) \xrightarrow{\phi_i \triangleright \tau_i} (\sigma_4, \overline{d}_4)$, or an execution $(\sigma_1, t_{main} : Main_2) \longrightarrow \cdots \longrightarrow (\sigma_3, \overline{d}_3) \xrightarrow{\phi_i \triangleright \tau_i} (\sigma_3, \mathsf{error})$. Both executions are witness for P_2 to the claim.

⟨5⟩2. Q.E.D.

⟨4⟩12. The above claim implies that we can replace the output of the second phase with a witness execution of P_2 to the claim, that either goes to the same store as P_1 or goes wrong, both from the same store.

⟨3⟩2. Q.E.D.

C.1 Proof of the soundness theorem for the QED steps

In the following we will prove a stronger version of Theorem 1 given in Section 4; the theorem below trivially implies the original version.

THEOREM 1 (Soundness). *Let $P_1, \mathsf{true} \dashrightarrow \cdots \dashrightarrow P_n, \mathcal{I}_n$ be a sequence of proof steps. Let* $\mathrm{V} = P_1.Var$ *and* $\mathrm{X} = P_n.Var \backslash P_1.Var$. *Then, the following holds:*

1. *For each $(\sigma_1, \sigma_2) \in Bad(P_1, \exists X. \mathcal{I}_n)$:*
   $\exists \sigma_3.\ (\sigma_1, \sigma_3) \in Bad|_{\mathrm{V}}(P_n, \mathcal{I}_n)$.
2. *For each $(\sigma_1, \sigma_2) \in Good(P_1, \exists X. \mathcal{I}_n)$:*
   (a) $\exists \sigma_3.\ (\sigma_1, \sigma_3) \in Bad|_{\mathrm{V}}(P_n, \mathcal{I}_n)$, or
   (b) $(\sigma_1, \sigma_2) \in Good|_{\mathrm{V}}(P_n, \mathcal{I}_n))$

PROOF SKETCH:Proof by induction on the length of the proof of the program.

⟨1⟩1. Base step. When there is no proof rule application, the claims hold ($X = V = \emptyset$).

⟨1⟩2. Inductive step. Assume that the claim holds for first n steps. We will prove that the claim still holds after the $n+1^{st}$ step. First we will prove the first claim about failing executions, and then the second claim about successful executions.

ASSUME: $P_1, \mathsf{true} \dashrightarrow \cdots \dashrightarrow P_n, \mathcal{I}_n \dashrightarrow P_{n+1}, \mathcal{I}_{n+1}$ be a proof of the program P_1.

ASSUME: Let $\mathrm{X} = P_n.Var \backslash P_1.Var$ and $\mathrm{Y} = P_{n+1}.Var \backslash P_1.Var$

ASSUME: Let $\mathrm{V} = P_1.Var$ and $\mathrm{W} = P_n.Var$

ASSUME: For each $(\sigma_1, \sigma_2) \in Bad(P_1, \exists \mathrm{X}. \mathcal{I}_n)$: $\exists \sigma_3.\ (\sigma_1, \sigma_3) \in Bad|_{\mathrm{V}}(P_n, \mathcal{I}_n)$

ASSUME: For each $(\sigma_1, \sigma_2) \in Good(P_1, \exists \mathrm{X}. \mathcal{I}_n)$: (i) $\exists \sigma_3.\ (\sigma_1, \sigma_3) \in Bad|_{\mathrm{V}}(P_n, \mathcal{I}_n)$ or (ii) $(\sigma_1, \sigma_2) \in Good|_{\mathrm{V}}(P_n, \mathcal{I}_n)$

⟨2⟩1. Prove Condition *1* of the theorem.

ASSUME: $(\sigma_1, \sigma_2) \in Bad(P_1, \exists \mathrm{Y}.\mathcal{I}_{n+1})$

PROVE: $\exists \sigma_3.\ (\sigma_1, \sigma_3) \in Bad|_{\mathrm{V}}(P_{n+1}, \mathcal{I}_{n+1})$

⟨3⟩1. $\sigma_1 \models \exists \mathrm{Y}.\ \mathcal{I}_{n+1}$

⟨3⟩2. By the proof rules, there are 2 cases: Either (i) $\mathrm{Y} = \mathrm{X}$ and $\mathcal{I}_{n+1} \Rightarrow \mathcal{I}_n$ or (ii) $\mathrm{Y} = \mathrm{X} \cup \{a\}$ and $\mathcal{I}_{n+1} = \mathcal{I}_n$

ASSUME: (i) $\mathrm{Y} = \mathrm{X}$ and $\mathcal{I}_{n+1} \Rightarrow \mathcal{I}_n$

⟨4⟩1. $\sigma_1 \models \exists \mathrm{Y}.\ \mathcal{I}_n$ by ⟨3⟩1 and assumption $\mathcal{I}_{n+1} \Rightarrow \mathcal{I}_n$.

⟨4⟩2. $\sigma_1 \models \exists \mathrm{X}.\mathcal{I}_n$ by ⟨4⟩1 and $\mathrm{X} = \mathrm{Y}$.

⟨4⟩3. $(\sigma_1, \sigma_2) \in Bad(P_1, \exists \mathrm{X}.\ \mathcal{I}_n)$ by definition of Bad.

⟨4⟩4. $\exists \sigma_3.\ (\sigma_1, \sigma_3) \in Bad|_{\mathrm{V}}(P_n, \mathcal{I}_n)$ by ⟨4⟩3 and inductive hypothesis.

⟨4⟩5. $\exists (\sigma_4, \sigma_5) \in Bad(P_n, \mathcal{I}_n)$ by definition of $Bad|_{\mathrm{V}}$ such that $\sigma_4|_{\mathrm{V}} = \sigma_1$, $\sigma_4 \models \mathcal{I}_{n+1}$ (by ⟨3⟩1 and $\mathrm{X} = \mathrm{Y}$), and $\sigma_5|_{\mathrm{V}} = \sigma_3$. Thus $(\sigma_4, \sigma_5) \in Bad(P_n, \mathcal{I}_{n+1})$.

⟨4⟩6. $\exists \sigma_6.\ (\sigma_4, \sigma_6) \in Bad|_{\mathrm{W}}(P_{n+1}, \mathcal{I}_{n+1})$ Lemma 7 and $\mathrm{Y} \backslash \mathrm{X} = \emptyset$.

⟨4⟩7. $(\sigma_1, \sigma_6|_{\mathrm{V}}) \in Bad|_{\mathrm{V}}(P_{n+1}, \mathcal{I}_{n+1})$ by ⟨4⟩6.

ASSUME: $\mathrm{Y} = \mathrm{X} \cup \{a\}$ and $\mathcal{I}_{n+1} = \mathcal{I}_n$

⟨4⟩1. $\sigma_1 \models \exists \mathrm{Y}.\ \mathcal{I}_n$ by ⟨3⟩1 and assumption $\mathcal{I}_{n+1} = \mathcal{I}_n$.

⟨4⟩2. Since $\mathcal{I}_n$ does not refer to a as a free variable, $\sigma_1 \models \exists \mathrm{X}.\ \mathcal{I}_n$ holds by ⟨4⟩1 and assumption $\mathrm{Y} = \mathrm{X} \cup \{a\}$.

⟨4⟩3. $(\sigma_1, \sigma_2) \in Bad(P_1, \exists \mathrm{X}.\ \mathcal{I}_n)$ by definition of Bad.

⟨4⟩4. $\exists \sigma_3.\ (\sigma_1, \sigma_3) \in Bad|_{\mathrm{V}}(P_n, \mathcal{I}_n)$ by ⟨4⟩3 and inductive hypothesis.

⟨4⟩5. $\exists (\sigma_4, \sigma_5) \in Bad(P_n, \mathcal{I}_n)$ by definition of $Bad|_{\mathrm{V}}$ such that $\sigma_4|_{\mathrm{V}} = \sigma_1$, $\sigma_4 \models \exists a.\ \mathcal{I}_{n+1}$ (by ⟨3⟩1 and $\mathrm{Y} \backslash \mathrm{X} = \{a\}$), and $\sigma_5|_{\mathrm{V}} = \sigma_3$. Thus $(\sigma_4, \sigma_5) \in Bad(P_n, \exists a.\ \mathcal{I}_{n+1})$.

⟨4⟩6. $\exists \sigma_6.\ (\sigma_4, \sigma_6) \in Bad|_{\mathrm{W}}(P_{n+1}, \mathcal{I}_{n+1})$ Lemma 7.

⟨4⟩7. $(\sigma_1, \sigma_6|_{\mathrm{V}}) \in Bad|_{\mathrm{V}}(P_{n+1}, \mathcal{I}_{n+1})$ by ⟨4⟩6.

⟨3⟩3. Q.E.D.

⟨2⟩2. Prove Condition *2* of the theorem.

ASSUME: $(\sigma_1, \sigma_2) \in Good(P_1, \exists \mathrm{Y}.\ \mathcal{I}_{n+1})$.

PROVE: Either (i) $\exists \sigma_3.\ (\sigma_1, \sigma_3) \in Bad|_{\mathrm{V}}(P_{n+1}, \mathcal{I}_{n+1})$ or (ii) $(\sigma_1, \sigma_2) \in Good|_{\mathrm{V}}(P_{n+1}, \mathcal{I}_{n+1})$

$\langle 3\rangle 1.$ $\sigma_1 \models \exists \mathrm{Y}.\mathcal{I}_{n+1}$
$\langle 3\rangle 2.$ By the proof rules, there are 2 cases: Either (i) $\mathrm{Y} = \mathrm{X}$ and $\mathcal{I}_{n+1} \Rightarrow \mathcal{I}_n$ or (ii) $\mathrm{Y} = \mathrm{X} \cup \{a\}$ and $\mathcal{I}_{n+1} = \mathcal{I}_n$
ASSUME: (i) $\mathrm{Y} = \mathrm{X}$ and $\mathcal{I}_{n+1} \Rightarrow \mathcal{I}_n$
$\langle 4\rangle 1.$ $\sigma_1 \models \exists \mathrm{Y}.\mathcal{I}_n$ by $\langle 3\rangle 1$ and assumption $\mathcal{I}_{n+1} \Rightarrow \mathcal{I}_n$.
$\langle 4\rangle 2.$ $\sigma_1 \models \exists \mathrm{X}.\mathcal{I}_n$ by $\langle 4\rangle 1$ and $\mathrm{X} = \mathrm{Y}$.
$\langle 4\rangle 3.$ $(\sigma_1, \sigma_2) \in \mathit{Good}(P_1, \exists \mathrm{X}.\mathcal{I}_n)$ by $\langle 4\rangle 2$ and definition of *Good*
$\langle 4\rangle 4.$ (i) $\exists \sigma_3.\ (\sigma_1, \sigma_3) \in \mathit{Bad}|_{\mathrm{V}}(P_n, \mathcal{I}_n)$ or (ii) $(\sigma_1, \sigma_2) \in \mathit{Good}|_{\mathrm{V}}(P_n, \mathcal{I}_n)$ by the inductive hypothesis.
ASSUME: $\exists \sigma_3.\ (\sigma_1, \sigma_3) \in \mathit{Bad}|_{\mathrm{V}}(P_n, \mathcal{I}_n)$
PROVE: $\exists \sigma_6.\ (\sigma_1, \sigma_6) \in \mathit{Bad}|_{\mathrm{V}}(P_{n+1}, \mathcal{I}_{n+1})$
$\langle 5\rangle 1.$ $\exists (\sigma_4, \sigma_5) \in \mathit{Bad}(P_n, \mathcal{I}_n)$ such that $\sigma_4|_{\mathrm{V}} = \sigma_1$, $\sigma_4 \models \mathcal{I}_{n+1}$ (by $\langle 3\rangle 1$ and $\mathrm{X} = \mathrm{Y}$) and $\sigma_5|_{\mathrm{V}} = \sigma_3$ by definition of $\mathit{Bad}|_{\mathrm{V}}$. Thus $(\sigma_4, \sigma_5) \in \mathit{Bad}(P_n, \mathcal{I}_{n+1})$.
$\langle 5\rangle 2.$ $\exists \sigma_6.\ (\sigma_4, \sigma_6) \in \mathit{Bad}|_{\mathrm{W}}(P_{n+1}, \mathcal{I}_{n+1})$ by Lemma 7 and $\mathrm{X} = \mathrm{Y}$.
$\langle 5\rangle 3.$ $(\sigma_1, \sigma_6|_{\mathrm{V}}) \in \mathit{Bad}|_{\mathrm{V}}(P_{n+1}, \mathcal{I}_{n+1})$ by $\langle 5\rangle 2$ and and the fact that $\mathrm{V} \subseteq \mathrm{W}$.
$\langle 5\rangle 4.$ Q.E.D.
ASSUME: $(\sigma_1, \sigma_2) \in \mathit{Good}|_{\mathrm{V}}(P_n, \mathcal{I}_n)$
$\langle 5\rangle 1.$ $\exists (\sigma_4, \sigma_5) \in \mathit{Good}(P_n, \mathcal{I}_n)$ such that $\sigma_4|_{\mathrm{V}} = \sigma_1$, $\sigma_4 \models \mathcal{I}_{n+1}$ (by $\langle 3\rangle 1$ and $\mathrm{X} = \mathrm{Y}$), and $\sigma_5|_{\mathrm{V}} = \sigma_3$ by definition of $\mathit{Good}|_{\mathrm{V}}$. Thus $(\sigma_4, \sigma_5) \in \mathit{Good}(P_n, \mathcal{I}_{n+1})$.
$\langle 5\rangle 2.$ By Lemma 7 and $\langle 5\rangle 1$, either $\exists \sigma_6.\ (\sigma_4, \sigma_6) \in \mathit{Bad}|_{\mathrm{W}}(P_{n+1}, \mathcal{I}_{n+1})$ or $(\sigma_4, \sigma_5) \in \mathit{Good}|_{\mathrm{W}}(P_{n+1}, \mathcal{I}_{n+1})$.
ASSUME: $\exists \sigma_6.\ (\sigma_4, \sigma_6) \in \mathit{Bad}|_{\mathrm{W}}(P_{n+1}, \mathcal{I}_{n+1})$
$\langle 6\rangle 1.$ $(\sigma_1, \sigma_6|_{\mathrm{V}}) \in \mathit{Bad}|_{\mathrm{V}}(P_{n+1}, \mathcal{I}_{n+1})$ by $\langle 5\rangle 1$ and the fact that $\mathrm{V} \subseteq \mathrm{W}$
$\langle 6\rangle 2.$ Q.E.D.
ASSUME: $(\sigma_4, \sigma_5) \in \mathit{Good}|_{\mathrm{W}}(P_{n+1}, \mathcal{I}_{n+1})$.
$\langle 6\rangle 1.$ $(\sigma_1, \sigma_2) \in \mathit{Good}|_{\mathrm{V}}(P_{n+1}, \mathcal{I}_{n+1})$ by $\langle 5\rangle 1$ and the fact that $\mathrm{V} \subseteq \mathrm{W}$
$\langle 6\rangle 2.$ Q.E.D.
$\langle 5\rangle 3.$ Q.E.D.
$\langle 4\rangle 5.$ Q.E.D.
ASSUME: $\mathrm{Y} = \mathrm{X} \cup \{a\}$ and $\mathcal{I}_{n+1} = \mathcal{I}_n$
$\langle 4\rangle 1.$ $\sigma_1 \models \exists \mathrm{Y}.\mathcal{I}_n$ by $\langle 3\rangle 1$ and assumption $\mathcal{I}_{n+1} = \mathcal{I}_n$.
$\langle 4\rangle 2.$ Since $\mathcal{I}_n$ does not refer to a as a free variable, $\sigma_1 \models \exists \mathrm{X}.\ \mathcal{I}_n$ holds by $\langle 4\rangle 1$ and assumption $\mathrm{Y} = \mathrm{X} \cup \{a\}$.
$\langle 4\rangle 3.$ $(\sigma_1, \sigma_2) \in \mathit{Good}(P_1, \exists \mathrm{X}.\mathcal{I}_n)$ by $\langle 4\rangle 2$ and definition of *Good*
$\langle 4\rangle 4.$ (i) $\exists \sigma_3.\ (\sigma_1, \sigma_3) \in \mathit{Bad}|_{\mathrm{V}}(P_n, \mathcal{I}_n)$ or (ii) $(\sigma_1, \sigma_2) \in \mathit{Good}|_{\mathrm{V}}(P_n, \mathcal{I}_n)$ by the inductive hypothesis.
ASSUME: $\exists \sigma_3.\ (\sigma_1, \sigma_3) \in \mathit{Bad}|_{\mathrm{V}}(P_n, \mathcal{I}_n)$
PROVE: $\exists \sigma_6.\ (\sigma_1, \sigma_6) \in \mathit{Bad}|_{\mathrm{V}}(P_{n+1}, \mathcal{I}_{n+1})$
$\langle 5\rangle 1.$ $\exists (\sigma_4, \sigma_5) \in \mathit{Bad}(P_n, \mathcal{I}_n)$ such that $\sigma_4|_{\mathrm{V}} = \sigma_1$, $\sigma_4 \models \exists a.\ \mathcal{I}_{n+1}$ (by $\langle 3\rangle 1$ and $\mathrm{Y}\backslash\mathrm{X} = \{a\}$), and $\sigma_5|_{\mathrm{V}} = \sigma_3$ by definition of $\mathit{Bad}|_{\mathrm{V}}$. Thus $(\sigma_4, \sigma_5) \in \mathit{Bad}(P_n, \exists a.\ \mathcal{I}_{n+1})$.
$\langle 5\rangle 2.$ $\exists \sigma_6.\ (\sigma_4, \sigma_6) \in \mathit{Bad}|_{\mathrm{W}}(P_{n+1}, \mathcal{I}_{n+1})$ by Lemma 7 and $\langle 5\rangle 1$.
$\langle 5\rangle 3.$ $(\sigma_1, \sigma_6|_{\mathrm{V}}) \in \mathit{Bad}|_{\mathrm{V}}(P_{n+1}, \mathcal{I}_{n+1})$ by $\langle 5\rangle 2$ and the fact that $\mathrm{V} \subseteq \mathrm{W}$.
$\langle 5\rangle 4.$ Q.E.D.
ASSUME: $(\sigma_1, \sigma_2) \in \mathit{Good}|_{\mathrm{V}}(P_n, \mathcal{I}_n)$
$\langle 5\rangle 1.$ $\exists (\sigma_4, \sigma_5) \in \mathit{Good}(P_n, \mathcal{I}_n)$ such that $\sigma_4|_{\mathrm{V}} = \sigma_1$, $\sigma_4 \models \exists a.\ \mathcal{I}_{n+1}$ (by $\langle 3\rangle 1$ and $\mathrm{Y}\backslash\mathrm{X} = \{a\}$), and $\sigma_5|_{\mathrm{V}} = \sigma_3$ by definition of $\mathit{Good}|_{\mathrm{V}}$. Thus $(\sigma_4, \sigma_5) \in \mathit{Bad}(P_n, \exists a.\ \mathcal{I}_{n+1})$.
$\langle 5\rangle 2.$ By Lemma 7 and $\langle 5\rangle 1$, either $\exists \sigma_6.\ (\sigma_4, \sigma_6) \in \mathit{Bad}|_{\mathrm{W}}(P_{n+1}, \mathcal{I}_{n+1})$ or $(\sigma_4, \sigma_5) \in \mathit{Good}|_{\mathrm{W}}(P_{n+1}, \mathcal{I}_{n+1})$.
ASSUME: $\exists \sigma_6.\ (\sigma_4, \sigma_6) \in \mathit{Bad}|_{\mathrm{W}}(P_{n+1}, \mathcal{I}_{n+1})$
$\langle 6\rangle 1.$ $(\sigma_1, \sigma_6|_{\mathrm{V}}) \in \mathit{Bad}|_{\mathrm{V}}(P_{n+1}, \mathcal{I}_{n+1})$ by $\langle 5\rangle 1$ and the fact that $\mathrm{V} \subseteq \mathrm{W}$
$\langle 6\rangle 2.$ Q.E.D.
ASSUME: $(\sigma_4, \sigma_5) \in \mathit{Good}|_{\mathrm{W}}(P_{n+1}, \mathcal{I}_{n+1})$.
$\langle 6\rangle 1.$ $(\sigma_1, \sigma_2) \in \mathit{Good}|_{\mathrm{V}}(P_{n+1}, \mathcal{I}_{n+1})$ by $\langle 5\rangle 1$ and the fact that $\mathrm{V} \subseteq \mathrm{W}$
$\langle 6\rangle 2.$ Q.E.D.
$\langle 5\rangle 3.$ Q.E.D.
$\langle 4\rangle 5.$ Q.E.D.
$\langle 2\rangle 3.$ Q.E.D.
$\langle 1\rangle 3.$ Q.E.D.

C.2 Proof of the checks for the simulation relation

LEMMA 8. *Let $\mathcal{I}$ be a store predicate. $\mathcal{I} \vdash \phi_1 \triangleright \tau_1 \preceq \phi_2 \triangleright \tau_2$ iff (i) $\models \mathcal{I} \wedge \phi_2 \Rightarrow \phi_1$, and (ii) $\models \mathcal{I} \wedge \phi_2 \wedge \tau_1 \Rightarrow \tau_2$.*

PROOF SKETCH:Prove each direction of *iff* separately.
$\langle 1\rangle 1.$ If $\mathcal{I} \vdash \phi_1 \triangleright \tau_1 \preceq \phi_2 \triangleright \tau_2$ then (i) $\models \mathcal{I} \wedge \phi_2 \Rightarrow \phi_1$, and (ii) $\models \mathcal{I} \wedge \phi_2 \wedge \tau_1 \Rightarrow \tau_2$
ASSUME: $\forall t \in \mathit{Tid}.\ \mathit{Bad}(t, \phi_1 \triangleright \tau_1, \mathcal{I}) \subseteq \mathit{Bad}(t, \phi_2 \triangleright \tau_2, \mathcal{I})$
PROVE: $\models \mathcal{I} \wedge \phi_2 \Rightarrow \phi_1$
$\langle 3\rangle 1.$ Take the contrapositive of the claim $\neg\phi_1 \Rightarrow \neg\mathcal{I} \vee \phi_2$
$\langle 3\rangle 2.$ Assume a failing execution $(\sigma_1, t : \phi_1 \triangleright \tau_1) \longrightarrow (\sigma_2, \mathsf{error})$.
$\langle 3\rangle 3.$ By $\langle 3\rangle 2$, $\sigma_1 \models \neg\phi_1$ and $\sigma_1 \models \mathcal{I}$
$\langle 3\rangle 4.$ By $\langle 3\rangle 2$ and assumption there is a failing execution $(\sigma_1, t : \phi_2 \triangleright \tau_2) \longrightarrow (\sigma_3, \mathsf{error})$
$\langle 3\rangle 5.$ By $\langle 3\rangle 4$, $\sigma_1 \models \neg\phi_2$ and $\sigma_1 \models \mathcal{I}$
$\langle 3\rangle 6.$ For any store $\sigma_1 \in \mathit{Stores}$, if $\sigma \models \neg\phi_1$ then either $\sigma \models \neg\mathcal{I}$, or by $\langle 3\rangle 3$ and $\langle 3\rangle 5$, $\sigma \models \mathcal{I} \wedge \neg\phi_2$.

$\langle 3\rangle 7$. By $\langle 3\rangle 6$, $\models \neg\phi_1 \Rightarrow (\neg\mathcal{I} \vee \neg\phi_2)$, thus $\models \mathcal{I} \wedge \phi_2 \Rightarrow \phi_1$
$\langle 3\rangle 8$. Q.E.D.
ASSUME: $\forall t \in Tid.\ Good(t, \phi_1 \triangleright \tau_1, \mathcal{I} \wedge \phi_2) \subseteq Good(t, \phi_2 \triangleright \tau_2, \mathcal{I})$
PROVE: $\models \mathcal{I} \wedge \phi_2 \wedge \tau_1 \Rightarrow \tau_2$
$\langle 3\rangle 1$. Assume a successful execution $(\sigma_1, t : \phi_1 \triangleright \tau_1) \longrightarrow (\sigma_2, \mathsf{skip})$.
$\langle 3\rangle 2$. By $\langle 3\rangle 1$, $\sigma_1 \models \mathcal{I} \wedge \phi_2$, $\sigma_1 \models \phi_1$ and $(\sigma_1, \sigma_2) \models \tau_1$.
$\langle 3\rangle 3$. By $\langle 3\rangle 1$ and assumption there is a successful execution $(\sigma_1, t : \phi_2 \triangleright \tau_2) \longrightarrow (\sigma_2, \mathsf{skip})$
$\langle 3\rangle 4$. By $\langle 3\rangle 3$, $\sigma_1 \models \phi_2$, $\sigma_1 \models \mathcal{I}$, and $(\sigma_1, \sigma_2) \models \tau_2$.
$\langle 3\rangle 5$. For any stores $\sigma_1, \sigma_2 \in Stores$, if $(\sigma_1, \sigma_2) \models \mathcal{I} \wedge \phi_2 \wedge \tau_1$ then $(\sigma_1, \sigma_2) \models \mathcal{I} \wedge \phi_2 \wedge \tau_2$
$\langle 3\rangle 6$. By $\langle 3\rangle 5$, $(\mathcal{I} \wedge \phi_2 \wedge \tau_1) \Rightarrow \tau_2$
$\langle 3\rangle 7$. Q.E.D.
$\langle 2\rangle 1$. Q.E.D.
$\langle 1\rangle 2$. If (i) $\models \mathcal{I} \wedge \phi_2 \Rightarrow \phi_1$, and (ii) $\models \mathcal{I} \wedge \phi_2 \wedge \tau_1 \Rightarrow \tau_2$ then $\mathcal{I} \vdash \phi_1 \triangleright \tau_1 \preceq \phi_2 \triangleright \tau_2$.
PROOF SKETCH:Assume the conditions and show the conditions in the definition of simulation; for any proper execution in one set, prove that there is an execution in the other.
ASSUME: $\models \mathcal{I} \wedge \phi_2 \Rightarrow \phi_1$
PROVE: $\forall t \in Tid.\ Bad(t, \phi_1 \triangleright \tau_1, \mathcal{I}) \subseteq Bad(t, \phi_2 \triangleright \tau_2, \mathcal{I})$
ASSUME: A failing execution $(\sigma_1, t : \phi_1 \triangleright \tau_1) \longrightarrow (\sigma_2, \mathsf{error})$ and $(\sigma_1, \sigma_2) \in Bad(t, \phi_1 \triangleright \tau_1, \mathcal{I})$
PROVE: $(\sigma_1, \sigma_2) \in Bad(t, \phi_2 \triangleright \tau_2, \mathcal{I})$
$\langle 4\rangle 1$. Taking the contrapositive of the assumption yields $\neg\phi_1 \Rightarrow \neg\mathcal{I} \vee \neg\phi_2$
$\langle 4\rangle 2$. By assumption there is an execution, so it must be that $\sigma_1 \models \mathcal{I}\neg\phi_1$
$\langle 4\rangle 3$. By $\langle 4\rangle 2$, $\sigma_1 \models \neg\phi_1$, so $\sigma_1 \models \neg\phi_2$ since $\sigma_1 \models \mathcal{I}$
$\langle 4\rangle 4$. By ATOMIC, there is an execution $(\sigma_1, t : \phi_2 \triangleright \tau_2) \longrightarrow (\sigma_2, \mathsf{error})$
$\langle 4\rangle 5$. By $\langle 4\rangle 4$ and the definition of Bad, $(\sigma_1, \sigma_2) \in Bad(t, \phi_2 \triangleright \tau_2, \mathcal{I})$
$\langle 4\rangle 6$. Q.E.D.
ASSUME: $\models \mathcal{I} \wedge \phi_2 \wedge \tau_1 \Rightarrow \tau_2$
PROVE: $\forall t \in Tid.\ Good(t, \phi_1 \triangleright \tau_1, \mathcal{I} \wedge \phi_2) \subseteq Good(t, \phi_2 \triangleright \tau_2, \mathcal{I})$
ASSUME: A successful execution $(\sigma_1, t : \phi_1 \triangleright \tau_1) \longrightarrow (\sigma_2, \mathsf{skip})$ and $(\sigma_1, \sigma_2) \in Good(t, \phi_1 \triangleright \tau_1, \mathcal{I} \wedge \phi_2)$
PROVE: $(\sigma_1, \sigma_2) \in Good(t, Good\phi_2\tau_2, \mathcal{I})$
$\langle 4\rangle 1$. By ATOMIC, $(\sigma_1, \sigma_2) \models \phi_1 \wedge \phi_2 \wedge \tau_1$
$\langle 4\rangle 2$. By $\langle 4\rangle 1$, $\sigma_1 \models \phi_2$, so ATOMIC applies.
$\langle 4\rangle 3$. By assumption and $\langle 4\rangle 2$, $(\sigma_1, \sigma_2) \models \tau_2$.
$\langle 4\rangle 4$. By ATOMIC, there is an execution $(\sigma_1, t : \phi_2 \triangleright \tau_2) \longrightarrow (\sigma_2, \mathsf{skip})$
$\langle 4\rangle 5$. By $\langle 4\rangle 5$ and the definition of $Good$, $(\sigma_1, \sigma_2) \in Good(t, \phi_2 \triangleright \tau_2, \mathcal{I})$
$\langle 4\rangle 6$. Q.E.D.
$\langle 2\rangle 1$. Q.E.D.
$\langle 1\rangle 3$. Q.E.D.

C.3 Some properties of movers

The following lemmas states that right and left movers preserve their mover types after composing statements via REDUCE-SEQUENTIAL and REDUCE-CHOICE rules rules.

LEMMA 9. *Let $P_1, \mathcal{I}_1 \longrightarrow P_2[(\alpha_1; \alpha_2) \mapsto \alpha_1 \circ \alpha_2], \mathcal{I}_2$ be a proof step by* REDUCE-SEQUENTIAL. *The following holds:*

1. *If $P_1, \mathcal{I}_1 \vdash \alpha : \mathbb{R}$ then $P_2, \mathcal{I}_2 \vdash \alpha : \mathbb{R}$.*
2. *If $P_1, \mathcal{I}_1 \vdash \alpha : \mathbb{L}$ then $P_2, \mathcal{I}_2 \vdash \alpha : \mathbb{L}$.*

PROOF SKETCH:We first prove that the $\preceq$ relation is transitive. Then we prove each condition by moving α by only one action to the right or left in two steps and concluding by transitivity the desired claim. We make use of the fact that our definition of $\circ$ is associative, whose proof is straightforward.

PROVE: The simulation relation $\preceq$ is transitive.
ASSUME: $a \triangleright x$, $b \triangleright y$ and $c \triangleright z$ be gated actions such that $a \triangleright x \preceq b \triangleright y$ and $b \triangleright y \preceq c \triangleright z$.
PROVE: $a \triangleright x \preceq c \triangleright z$
$\langle 3\rangle 1$. $c \Rightarrow b$ and $b \Rightarrow a$ by assumption and Lemma 8, thus $c \Rightarrow a$.
$\langle 3\rangle 2$. $c \wedge y \Rightarrow z$ and $b \wedge x \Rightarrow y$ by assumption and Lemma 8, thus $c \wedge x \Rightarrow z$.
$\langle 3\rangle 3$. $b \triangleright y \preceq c \triangleright z$ by $\langle 3\rangle 1$, $\langle 3\rangle 2$ and Lemma 8.
$\langle 3\rangle 4$. Q.E.D.
ASSUME: $t \neq u$, $\alpha[t/tid] = \phi \triangleright \tau$, $\alpha_1[u/tid] = \phi_1 \triangleright \tau_1$, and $\alpha_2[u/tid] = \phi_2 \triangleright \tau_2$.
PROVE: Condition *1*
ASSUME: $(\mathsf{true} \triangleright (\phi \wedge \tau) \circ \phi_1 \triangleright \tau_1) \preceq (\phi_1 \triangleright \tau_1 \circ \mathsf{true} \triangleright (\phi \wedge \tau))$
ASSUME: $\phi_1 \Rightarrow \mathsf{wp}(\phi \wedge \tau, \phi_1)$ by Lemma 8
ASSUME: $((\phi_1 \wedge \tau) \circ \tau_1) \Rightarrow (\tau_1 \circ \tau)$, by Lemma 8
ASSUME: $(\mathsf{true} \triangleright (\phi \wedge \tau) \circ \phi_2 \triangleright \tau_2) \preceq (\phi_2 \triangleright \tau_2 \circ \mathsf{true} \triangleright (\phi \wedge \tau))$
ASSUME: $\phi_2 \Rightarrow \mathsf{wp}(\phi \wedge \tau, \phi_2)$, by Lemma 8
ASSUME: $((\phi_2 \wedge \tau) \circ \tau_2) \Rightarrow (\tau_2 \circ \tau)$, by Lemma 8
PROVE: $(\mathsf{true} \triangleright \phi \wedge \tau \circ \phi_1 \triangleright \tau_1 \circ \phi_2 \triangleright \tau_2) \preceq (\phi_1 \triangleright \tau_1 \circ \mathsf{true} \triangleright \phi \wedge \tau \circ \phi_2 \triangleright \tau_2)$

⟨2⟩1. The conditions $(\phi_1 \wedge \mathsf{wp}(\tau_1 \circ (\phi \wedge \tau), \phi_2)) \Rightarrow (\mathsf{wp}(\phi \wedge \tau, \phi_1) \wedge \mathsf{wp}((\phi \wedge \tau) \circ \tau_1, \phi_2))$ and $(\phi_1 \wedge \mathsf{wp}(\tau_1 \circ (\phi \wedge \tau), \phi_2) \wedge ((\phi \wedge \tau) \circ \tau_1 \circ \tau_2)) \Rightarrow (\tau_1 \circ (\phi \wedge \tau) \circ \tau_2))$ are trivially entailed by the assumptions.

PROVE: $(\phi_1 \triangleright \tau_1 \circ \mathsf{true} \triangleright \phi \wedge \tau \circ \phi_2 \triangleright \tau_2) \preceq (\phi_1 \triangleright \tau_1 \circ \phi_2 \triangleright \tau_2 \circ \mathsf{true} \triangleright \phi \wedge \tau)$

⟨2⟩1. The conditions $(\phi_1 \wedge \mathsf{wp}(\tau_1, \phi_2)) \Rightarrow (\phi_1 \wedge \mathsf{wp}(\tau_1 \circ (\phi \wedge \tau), \phi_2))$ and $(\phi_1 \wedge \mathsf{wp}(\tau_1, \phi_2) \wedge (\tau_1 \circ (\phi \wedge \tau) \circ \tau_2)) \Rightarrow (\tau_1 \circ \tau_2 \circ (\phi \wedge \tau)))$ are trivially entailed by the assumptions.

⟨1⟩1. By transitivity of $\preceq$, associativity of $\circ$, and the simulation relations proved above, $(\mathsf{true} \triangleright (\phi \wedge \tau) \circ \phi_1 \triangleright \tau_1 \circ \phi_2 \triangleright \tau_2) \preceq (\phi_1 \triangleright \tau_1 \circ \phi_2 \triangleright \tau_2 \circ \mathsf{true} \triangleright (\phi \wedge \tau))$ holds, thus α is a right-mover with respect to $\alpha_1 \circ \alpha_2$.

⟨1⟩2. Q.E.D.

PROVE: Condition *2*

ASSUME: $(\phi_1 \triangleright \tau_1 \circ \phi \triangleright \tau) \preceq (\phi \triangleright \tau \circ \phi_1 \triangleright \tau_1)$

ASSUME: $(\phi \wedge \mathsf{wp}(\tau, \phi_1)) \Rightarrow (\phi_1 \wedge \mathsf{wp}(\tau_1, \phi))$ by Lemma 8

ASSUME: $(\phi \wedge \mathsf{wp}(\tau, \phi_1) \wedge (\tau_1 \circ \tau)) \Rightarrow (\tau \circ \tau_1)$, by Lemma 8

ASSUME: $(\phi_2 \triangleright \tau_2 \circ \phi \triangleright \tau) \preceq (\phi \triangleright \tau \circ \phi_2 \triangleright \tau_2)$

ASSUME: $(\phi \wedge \mathsf{wp}(\tau, \phi_2)) \Rightarrow (\phi_2 \wedge \mathsf{wp}(\tau_2, \phi))$ by Lemma 8

ASSUME: $(\phi \wedge \mathsf{wp}(\tau, \phi_2) \wedge (\tau_2 \circ \tau)) \Rightarrow (\tau \circ \tau_2)$, by Lemma 8

PROVE: $(\phi_1 \triangleright \tau_1 \circ \phi_2 \triangleright \tau_2 \circ \phi \triangleright \tau) \preceq (\phi_1 \triangleright \tau_1 \circ \phi \triangleright \tau \circ \phi_2 \triangleright \tau_2)$

⟨2⟩1. The conditions $(\phi_1 \wedge \mathsf{wp}(\tau_1, \phi) \wedge \mathsf{wp}(\tau_1 \circ \tau, \phi_2)) \Rightarrow (\phi_1 \wedge \mathsf{wp}(\tau_1, \phi_2) \wedge \mathsf{wp}(\tau_1 \circ \tau_2, \phi))$ and $(\phi_1 \wedge \mathsf{wp}(\tau_1, \phi) \wedge \mathsf{wp}(\tau_1 \circ \tau, \phi_2) \wedge (\tau_1 \circ \tau_2 \circ \tau)) \Rightarrow (\tau_1 \circ \tau \circ \tau_2)$ are trivially entailed by the assumptions.

PROVE: $(\phi_1 \triangleright \tau_1 \circ \phi \triangleright \tau \circ \phi_2 \triangleright \tau_2) \preceq (\phi \triangleright \tau \circ \phi_1 \triangleright \tau_1 \circ \phi_2 \triangleright \tau_2)$

⟨2⟩1. The conditions $(\phi \wedge \mathsf{wp}(\tau, \phi_1) \wedge \mathsf{wp}(\tau \circ \tau_1, \phi_2)) \Rightarrow (\phi_1 \wedge \mathsf{wp}(\tau_1, \phi) \wedge \mathsf{wp}(\tau_1 \circ \tau, \phi_2))$ and $(\phi \wedge \mathsf{wp}(\tau, \phi_1) \wedge \mathsf{wp}(\tau \circ \tau_1, \phi_2)(\tau_1 \circ \tau \circ \tau_2)) \Rightarrow (\tau \circ \tau_1 \circ \tau_2)$ are trivially entailed by the assumptions.

⟨1⟩3. By transitivity of $\preceq$, associativity of $\circ$, and the simulation relations proved above, $(\phi_1 \triangleright \tau_1 \circ \phi_2 \triangleright \tau_2 \circ \phi \triangleright \tau) \preceq (\phi \triangleright \tau \circ \phi_1 \triangleright \tau_1 \circ \phi_1 \triangleright \tau_2)$ holds, thus α is a right-mover with respect to $\alpha_1 \circ \alpha_2$.

⟨1⟩4. Q.E.D.

LEMMA 10. *Let* $P_1, \mathcal{I}_1 \longrightarrow P_2[(\alpha_1 \Box \alpha_2) \mapsto \alpha_1 \oplus \alpha_2], \mathcal{I}_2$ *be a proof step by* REDUCE-CHOICE*. The following holds:*

1. *If* $P_1, \mathcal{I}_1 \vdash \alpha : \mathbb{R}$ *then* $P_2, \mathcal{I}_2 \vdash \alpha : \mathbb{R}$.
2. *If* $P_1, \mathcal{I}_1 \vdash \alpha : \mathbb{L}$ *then* $P_2, \mathcal{I}_2 \vdash \alpha : \mathbb{L}$.

ASSUME: $t \neq u$, $\alpha[t/tid] = \phi \triangleright \tau$, $\alpha_1[u/tid] = \phi_1 \triangleright \tau_1$, and $\alpha_2[u/tid] = \phi_2 \triangleright \tau_2$.

PROVE: Condition *1*

ASSUME: $(\mathsf{true} \triangleright (\phi \wedge \tau) \circ \phi_1 \triangleright \tau_1) \preceq (\phi_1 \triangleright \tau_1 \circ \mathsf{true} \triangleright (\phi \wedge \tau))$

ASSUME: $\phi_1 \Rightarrow \mathsf{wp}(\tau, \phi_1)$ by Lemma 8

ASSUME: $((\phi_1 \wedge \tau) \circ \tau_1) \Rightarrow (\tau_1 \circ \tau)$, by Lemma 8

ASSUME: $(\mathsf{true} \triangleright (\phi \wedge \tau) \circ \phi_2 \triangleright \tau_2) \preceq (\phi_2 \triangleright \tau_2 \circ \mathsf{true} \triangleright (\phi \wedge \tau))$

ASSUME: $\phi_2 \Rightarrow \mathsf{wp}(\tau, \phi_2)$, by Lemma 8

ASSUME: $((\phi_2 \wedge \tau) \circ \tau_2) \Rightarrow (\tau_2 \circ \tau)$, by Lemma 8

PROVE: $(\mathsf{true} \triangleright (\phi \wedge \tau) \circ (\phi_1 \wedge \phi_2 \triangleright \tau_1 \vee \phi_2)) \preceq ((\phi_1 \wedge \phi_2 \triangleright \tau_1 \vee \phi_2) \circ \mathsf{true} \triangleright (\phi \wedge \tau))$

⟨2⟩1. The conditions $(\phi_1 \wedge \phi_2) \Rightarrow \mathsf{wp}((\phi \wedge \tau), \phi_1 \wedge \phi_2)$ and $(\phi_1 \wedge \phi_2 \wedge ((\phi \circ \tau) \circ (\tau_1 \vee \tau_2))) \Rightarrow ((\tau_1 \vee \tau_2) \circ (\phi \wedge \tau))$ are trivially entailed by the assumptions, having the the following facts: $\mathsf{wp}(\tau_1 \vee \tau_2, \phi) = \mathsf{wp}(\tau_1, \phi) \wedge \mathsf{wp}(\tau_2, \phi)$, $\mathsf{wp}(\tau, \phi_1 \wedge \phi_2) = \mathsf{wp}(\tau, \phi_1) \wedge \mathsf{wp}(\tau, \phi_2)$, $\tau \circ (\tau_1 \vee \tau_2) = (\tau \circ \tau_1) \vee (\tau \circ \tau_2)$, and $(\tau_1 \vee \tau_2) \circ \tau = (\tau_1 \circ \tau) \vee (\tau_2 \circ \tau)$.

PROVE: Condition *2*

⟨2⟩1. The proof of this case is done similarly to the first one.

⟨1⟩1. Q.E.D.