Linearizability with Ownership Transfer

Alexey Gotsman and Hongseok Yang

[1] IMDEA Software Institute
[2] University of Oxford

Abstract. Linearizability is a commonly accepted notion of correctness for libraries of concurrent algorithms. Unfortunately, it assumes a complete isolation between a library and its client, with interactions limited to passing values of a given data type. This is inappropriate for common programming languages, where libraries and their clients can communicate via the heap, transferring the ownership of data structures, and can even run in a shared address space without any memory protection. In this paper, we present the first definition of linearizability that lifts this limitation and establish an Abstraction Theorem: while proving a property of a client of a concurrent library, we can soundly replace the library by its abstract implementation related to the original one by our generalisation of linearizability. We also prove that linearizability with ownership transfer can be derived from the classical one if the library does not access some of data structures transferred to it by the client.

1 Introduction

The architecture of concurrent software usually exhibits some forms of modularity. For example, concurrent algorithms are encapsulated in libraries and complex algorithms are often constructed using libraries of simpler ones. This lets developers benefit from ready-made libraries of concurrency patterns and high-performance concurrent data structures, such as `java.util.concurrent` for Java and Threading Building Blocks for C++. To simplify reasoning about concurrent software, we need to exploit the available modularity. In particular, in reasoning about a client of a concurrent library, we would like to abstract from the details of a particular library implementation. This requires an appropriate notion of library correctness.

Correctness of concurrent libraries is commonly formalised by *linearizability* [12], which fixes a certain correspondence between the library and its specification. The latter is usually just another library, but implemented atomically using an abstract data type. A good notion of linearizability should validate an *Abstraction Theorem* [10]: it is sound to replace a library with its specification in reasoning about its client.

The classical linearizability assumes a complete isolation between a library and its client, with interactions limited to passing values of a given data type as parameters or return values of library methods. This notion is not appropriate for low-level heap-manipulating languages, such as C/C++. There the library and the client run in a shared address space; thus, to prove the whole program correct, we need to verify that one of them does not corrupt the data structures used by the other. Type systems [6] and program logics [13] usually establish this using the concept of *ownership* of data structures by a program component. When verifying realistic programs, this ownership of data structures cannot be assigned statically; rather, it should be *transferred* between

the client and the library at calls to and returns from the latter. The times when ownership is transferred are not determined operationally, but set by the proof method: as O'Hearn famously put it, "ownership is in the eye of the asserter" [13]. However, ownership transfer reflects actual interactions between program components via the heap, e.g., alternating accesses to a shared area of memory. Such interactions also exist in high-level languages providing basic memory protection, such as Java.

For an example of ownership transfer between concurrent libraries and their clients consider a memory allocator accessible concurrently to multiple threads. We can think of the allocator as owning the blocks of memory on its free-list; in particular, it can store free-list pointers in them. Having allocated a block, a thread gets its exclusive ownership, which allows accessing it without interference from the other threads. When the thread frees the block, its ownership is returned to the allocator.

As another example, consider any container with concurrent access, such as a concurrent set from java.util.concurrent or Threading Building Blocks. A typical use of such a container is to store pointers to a certain type of data structures. However, when verifying a client of the container, we usually think of the latter as holding the ownership of the data structures whose addresses it stores [13]. Thus, when a thread inserts a pointer to a data structure into a container, its ownership is transferred from the thread to the container. When another thread removes a pointer from the container, it acquires the ownership of the data structure the pointer identifies. If the first thread tries to access a data structure after a pointer to it has been inserted into the container, this may result in a race condition. Unlike a memory allocator, the container code usually does not access the contents of the data structures its elements identify, but merely ferries their ownership between different threads. For this reason, correctness proofs for such containers [1, 7, 17] have so far established their classical linearizability, without taking ownership transfer into account.

We would like to use the notion of linearizability and, in particular, an Abstraction Theorem to reason about above libraries and their clients in isolation, taking into account only the memory that they own. When clients use the libraries to implement the ownership transfer paradigm, the correctness of the latter cannot be defined only in terms of passing pointers between the library and the client; we must also show that they perform ownership transfer correctly. So far, there has been no notion of linearizability that would allow this. In the case of concurrent containers, we cannot use classical linearizability established for them to validate an Abstraction Theorem that would be applicable to clients performing ownership transfer. This paper fills in these gaps.

Contributions. In this paper, we generalise linearizability to a setting where a library and its client execute in a shared address space, and boundaries between their data structures can change via ownership transfers (Section 3). Linearizability is usually defined in terms of *histories*, which are sequences of calls to and returns from a library in a given program execution, recording parameters and return values passed. To handle ownership transfer, histories also have to include descriptions of memory areas transferred. However, in this case, some histories cannot be generated by any pair of a client and a library. For example, a client that transfers an area of memory upon a call to a library not communicating with anyone else cannot then transfer the same area again before getting it back from the library upon a method return.

We propose a notion of *balancedness* that characterises those histories that treat ownership transfer correctly. We then define a *linearizability relation* between balanced

histories, matching histories of an implementation and a specification of a library (Section 3). We show that the proposed linearizability relation on histories is correct in the sense that it validates a Rearrangement Lemma (Lemma 13, Section 4): if a history H' linearizes another history H, and it can be produced by some execution of a library, then so can the history H. The need to consider ownership transfer makes the proof of the lemma highly non-trivial. This is because changing the history from H' to H requires moving calls and returns to different points in the computation. In the setting without ownership transfer, these actions are thread-local and can be moved easily; however, once they involve ownership transfer, they become global and the justification of their moves becomes subtle, in particular, relying on the fact that the histories involved are balanced (see the discussion in Section 4).

To lift the linearizability relation on histories to libraries and establish the Abstraction Theorem, we define a novel compositional semantics for a language with libraries that defines the denotation of a library or a client considered separately in an environment that communicates with the component correctly via ownership transfers (Section 6). To define such a semantics for a library, we generalise the folklore notion of its *most general client* to allow ownership transfers, which gives us a way to generate all possible library histories and lift the notion of linearizabiliy to libraries. We prove that our compositional semantics is sound and adequate with respect to the standard non-compositional semantics (Lemmas 16 and 17). This, together with the Rearrangement Lemma, allows us to establish the Abstraction Theorem (Theorem 19, Section 7).

To avoid having to prove the new notion of linearizability from scratch for libraries that do not access some of the data structures transferred to them, such as concurrent containers, we propose a *frame rule for linearizability* (Theorem 22, Section 8). It ensures the linearizability of such libraries with respect to a specification with ownership transfer given their linearizability with respect to a specification without one.

The Abstraction Theorem is not just a theoretical result: it enables compositional reasoning about complex concurrent algorithms that are challenging for existing verification methods (Section 7). We have also developed a logic, based on separation logic [14], for establishing our linearizability. Due to space constraints, the details of the logic are outside the scope of this paper. For the same reason, proofs of most theorems are given in Appendix B.

2 Footprints of States

Our results hold for a class of models of program states called *separation algebras* [5], which allow expressing the dynamic memory partitioning between libraries and clients.

Definition 1. *A **separation algebra** is a set Σ, together with a partial commutative, associative and cancellative operation $*$ on Σ and a unit element $\epsilon \in \Sigma$. Here unity, commutativity and associativity hold for the equality that means both sides are defined and equal, or both are undefined. The property of cancellativity says that for each $\theta \in \Sigma$, the function $\theta * \cdot : \Sigma \rightharpoonup \Sigma$ is injective.*

We think of elements of a separation algebra Σ as *portions* of program states and the $*$ operation as combining such portions. The partial states allow us to describe parts of the program state belonging to a library or the client. When the $*$-combination of two states is defined, we call them ***compatible***. We sometimes use a pointwise lifting $* : 2^\Sigma \times 2^\Sigma \to 2^\Sigma$ of $*$ to sets of states.

Elements of separation algebras are often defined using partial functions. We use the following notation: $g(x)\downarrow$ means that the function g is defined on x, $\mathsf{dom}(g)$ denotes the set of arguments on which g is defined, and $g[x : y]$ denotes the function that has the same value as g everywhere, except for x, where it has the value y. We also write $\_$ for an expression whose value is irrelevant and implicitly existentially quantified.

Below is an example separation algebra RAM:

$$\mathsf{Loc} = \{1, 2, \ldots\}; \qquad \mathsf{Val} = \mathbb{Z}; \qquad \mathsf{RAM} = \mathsf{Loc} \rightharpoonup_{fin} \mathsf{Val}.$$

A (partial) state in this model consists of a finite partial function from allocated memory locations to the values they store. The $*$ operation on RAM is defined as the disjoint function union $\uplus$, with the everywhere-undefined function $[\,]$ as its unit. Thus, the $*$ operation combines disjoint pieces of memory.

We define a partial operation $\setminus : \Sigma \times \Sigma \rightharpoonup \Sigma$, called ***state subtraction***, as follows: $\theta_2 \setminus \theta_1$ is a state in Σ such that $\theta_2 = (\theta_2 \setminus \theta_1) * \theta_1$; if such a state does not exist, $\theta_2 \setminus \theta_1$ is undefined. When reasoning about ownership transfer between a library and a client, we use the $*$ operation to express a state change for the component that is receiving the ownership of memory, and the $\setminus$ operation, for the one that is giving it up.

Our definition of linearizability uses a novel formalisation of a *footprint* of a state, which, informally, describes the amount of memory or permissions the state includes.

Definition 2. *A **footprint** of a state θ in a separation algebra Σ is the set of states $\delta(\theta) = \{\theta' \mid \forall \theta''. (\theta' * \theta'')\downarrow \Leftrightarrow (\theta * \theta'')\downarrow\}$.*

The function δ computes the equivalence class of states with the same footprint as θ. In the case of RAM, we have $\delta(\theta) = \{\theta' \mid \mathsf{dom}(\theta) = \mathsf{dom}(\theta')\}$ for every $\theta \in \mathsf{RAM}$. Thus, states with the same footprint contain the same memory cells.

Let $\mathcal{F}(\Sigma) = \{\delta(\theta) \mid \theta \in \Sigma\}$ be the set of footprints in a separation algebra Σ. We now lift the $*$ and $\setminus$ operations on Σ to $\mathcal{F}(\Sigma)$. First, we define the operation $\circ : \mathcal{F}(\Sigma) \times \mathcal{F}(\Sigma) \rightharpoonup \mathcal{F}(\Sigma)$ for adding footprints. Consider $l_1, l_2 \in \mathcal{F}(\Sigma)$ and $\theta_1, \theta_2 \in \Sigma$ such that $l_1 = \delta(\theta_1)$ and $l_2 = \delta(\theta_2)$. If $\theta_1 * \theta_2$ is defined, we let $l_1 \circ l_2 = \delta(\theta_1 * \theta_2)$; otherwise $l_1 \circ l_2$ is undefined. Choosing θ_1 and θ_2 differently does not lead to a different result (Appendix B). For RAM, $\circ$ is just a pointwise lifting of $*$. To define a subtraction operation on footprints, we use the following condition.

Definition 3. *The $*$ operation of a separation algebra Σ is **cancellative on footprints** when for all $\theta_1, \theta_2, \theta_1', \theta_2' \in \Sigma$, if $\theta_1 * \theta_2$ and $\theta_1' * \theta_2'$ are defined, then*

$$(\delta(\theta_1 * \theta_2) = \delta(\theta_1' * \theta_2') \wedge \delta(\theta_1) = \delta(\theta_1')) \Rightarrow \delta(\theta_2) = \delta(\theta_2').$$

For example, the $*$ operation on RAM satisfies this condition.

When $*$ of Σ is cancellative on footprints, we can define an operation $\setminus : \mathcal{F}(\Sigma) \times \mathcal{F}(\Sigma) \rightharpoonup \mathcal{F}(\Sigma)$ of ***footprint subtraction*** as follows. Consider $l_1, l_2 \in \mathcal{F}(\Sigma)$. If for some $\theta_1, \theta_2, \theta \in \Sigma$, we have $l_1 = \delta(\theta_1)$, $l_2 = \delta(\theta_2)$ and $\theta_2 = \theta_1 * \theta$, then we let $l_2 \setminus l_1 = \delta(\theta)$. When such $\theta_1, \theta_2, \theta$ do not exist, $l_2 \setminus l_1$ is undefined. Again, we can show that this definition is well-formed (Appendix B). We say that a footprint l_1 is ***smaller*** than l_2, written $l_1 \preceq l_2$, when $l_2 \setminus l_1$ is defined. In the rest of the paper, we fix a separation algebra Σ with the $*$ operation cancellative on footprints.

3 Linearizability with Ownership Transfer

In the following, we consider descriptions of computations of a library providing several methods to a multithreaded client. We fix the set ThreadID of thread identifiers and the set Method of method names. A good definition of linearizability has to allow replacing a concrete library implementation with its abstract version while keeping client behaviours reproducible. For this, it should require that the two libraries have similar client-observable behaviours. Such behaviours are recorded using *histories*, which we now define in our setting.

Definition 4. *An **interface action** ψ is an expression of the form $(t, \mathsf{call}\ m(\theta))$ or $(t, \mathsf{ret}\ m(\theta))$, where $t \in \mathsf{ThreadID}$, $m \in \mathsf{Method}$ and $\theta \in \Sigma$.*

An interface action records a call to or a return from a library method m by thread t. The component θ in $(t, \mathsf{call}\ m(\theta))$ specifies the part of the state transferred upon the call from the client to the library; θ in $(t, \mathsf{ret}\ m(\theta))$ is transferred in the other direction. For example, in the algebra RAM (Section 2), the annotation $\theta = [42 : 0]$ implies the transfer of the cell at the address 42 storing 0.

Definition 5. *A **history** H is a finite sequence of interface actions such that for every thread t, its projection $H|_t$ to actions by t is a sequence of alternating call and return actions over matching methods that starts from a call action.*

In the following, we use the standard notation for sequences: ε is the empty sequence, $\alpha(i)$ is the i-th element of a sequence α, and $|\alpha|$ is the length of α.

Not all histories make intuitive sense with respect to the ownership transfer reading of interface actions. For example, let $\Sigma = \mathsf{RAM}$ and consider the history

$$(1, \mathsf{call}\ m_1([10:0]))\,(2, \mathsf{call}\ m_2([10:0]))(2, \mathsf{ret}\ m_2([\,]))\,(1, \mathsf{ret}\ m_1([\,])).$$

The history is meant to describe *all* the interactions between the library and the client. According to the history, the cell at the address 10 was first owned by the client, and then transferred to the library by thread 1. However, before this state was transferred back to the client, it was again transferred from the client to the library, this time by thread 2. This is not consistent with the intuition of ownership transfer, as executing the second action requires the cell to be owned both by the library and by the client, which is impossible in RAM.

As we show in this paper, histories that do not respect the notion of ownership, such as the one above, cannot be generated by any program, and should not be taken into account when defining linearizability. We use the notion of footprints of states from Section 2 to characterise formally the set of histories that respect ownership.

A finite history H induces a partial function $[\![H]\!]^\sharp : \mathcal{F}(\Sigma) \rightharpoonup \mathcal{F}(\Sigma)$, which tracks how a computation with the history H changes the footprint of the library state:

$$\begin{array}{ll} [\![\varepsilon]\!]^\sharp l = l; & [\![H\psi]\!]^\sharp l = [\![H]\!]^\sharp l \circ \delta(\theta), \text{ if } \psi = (\_, \mathsf{call}\ \_(\theta)) \wedge ([\![H]\!]^\sharp l \circ \delta(\theta))\!\downarrow; \\ & [\![H\psi]\!]^\sharp l = [\![H]\!]^\sharp l \setminus \delta(\theta), \text{ if } \psi = (\_, \mathsf{ret}\ \_(\theta)) \wedge ([\![H]\!]^\sharp l \setminus \delta(\theta))\!\downarrow; \\ & [\![H\psi]\!]^\sharp l = \text{undefined}, \quad \text{otherwise.} \end{array}$$

Definition 6. *A history H is **balanced** from $l \in \mathcal{F}(\Sigma)$ if $[\![H]\!]^\sharp(l)$ is defined.*

Let $\mathsf{BHistory} = \{(l, H) \mid H \text{ is balanced from } l\}$ be the set of balanced histories and their initial footprints.

Definition 7. ***Linearizability*** *is a binary relation* $\sqsubseteq$ *on* BHistory *defined as follows:* $(l, H) \sqsubseteq (l', H')$ *holds iff (i)* $l' \preceq l$*; (ii)* $H|_t = H'|_t$ *for all* $t \in \mathsf{ThreadID}$*; and (iii) there exists a bijection* $\pi\colon \{1, \ldots, |H|\} \to \{1, \ldots, |H'|\}$ *such that for all* i *and* j*,*

$$H(i) = H'(\pi(i)) \wedge ((i < j \wedge H(i) = (\_, \mathsf{ret}\ \_) \wedge H(j) = (\_, \mathsf{call}\ \_)) \Rightarrow \pi(i) < \pi(j)).$$

A history H' linearizes a history H when it is a permutation of the latter preserving the order of actions within threads and non-overlapping method invocations. We additionally require that the initial footprint of H' be smaller than that of H, which is a standard requirement in data refinement [9]. It does not pose problems in practice, as the abstract library generating H' usually represents some of the data structures of the concrete library as abstract data types, which do not use the heap.

Definition 7 treats parts of memory whose ownership is passed between the library and the client in the same way as parameters and return values in the classical definition [12]: they are required to be the same in the two histories. In fact, the setting of the classical definition can be modelled in ours if we pass parameters and return values via the heap. The novelty of our definition lies in restricting the histories considered to balanced ones. This restriction is required for our notion of linearizability to be correct in the sense of the Rearrangement Lemma established in the next section.

4 Rearrangement Lemma

Intuitively, the Rearrangement Lemma says that, if $H \sqsubseteq H'$, then every execution trace of a library producing H' can be transformed into another trace of the same library that differs from the original one only in interface actions and produces H, instead of H'. This property is the key component for establishing the correctness of linearizability *on libraries*, formulated by the Abstraction Theorem in Section 7.

Primitive commands. We first define a set of primitive commands that clients and libraries can execute to change the memory atomically. Consider the set $2^{\Sigma} \cup \{\top\}$ of subsets of Σ with a special element $\top$ used to denote an error state, resulting, e.g., from dereferencing an invalid pointer. We assume a collection of primitive commands PComm and an interpretation of every $c \in \mathsf{PComm}$ as a transformer $f_c^t : \Sigma \to (2^{\Sigma} \cup \{\top\})$, which maps pre-states to states obtained when thread $t \in \mathsf{ThreadID}$ executes c from a pre-state. The fact that our transformers are parameterised by t allows atomic accesses to areas of memory indexed by thread identifiers. This idealisation simplifies the setting in that it lets us do without special thread-local or method-local storage for passing method parameters and return values. For our results to hold, we need to place some standard restrictions on the transformers f_c^t (see Appendix A).

Traces. We record information about a program execution, including internal actions by components, using *traces*.

Definition 8. *An* ***action*** φ *is either an interface action or an expression of the form* (t, c)*, where* $t \in \mathsf{ThreadID}$ *and* $c \in \mathsf{PComm}$*. We denote the set of all actions by* Act*.*

Definition 9. *A* ***trace*** τ *is a finite sequence of actions such that its projection* $\mathsf{history}(\tau)$ *to interface actions is a history. A trace* η *is a* ***client trace****, if*

$$\forall i, j, t, c.\, i < j \wedge \eta(i) = (t, \mathsf{call}\ \_) \wedge \eta(j) = (t, c) \Rightarrow \exists k.\, i < k < j \wedge \eta(k) = (t, \mathsf{ret}\ \_).$$

*A trace ξ is a **library trace**, if*

$$\forall i, t, c.\, \xi(i) = (t, c) \Rightarrow \exists j.\, j < i \wedge \xi(j) = (t, \mathsf{call}\ \_) \wedge \neg\exists k.\, i < k < j \wedge \xi(k) = (t, \mathsf{ret}\ \_).$$

In other words, a thread in a client trace cannot execute actions inside a library method, and in a library trace, outside it. We denote the set of all traces by Trace. In the following, η denotes client traces, ξ, library traces, and τ, arbitrary ones.

In this section, we are concerned with library traces only. For a library trace ξ, we define a function $[\![\xi]\!]_{\mathsf{lib}} : 2^{\Sigma} \to (2^{\Sigma} \cup \{\top\})$ that ***evaluates*** ξ, computing the state of the memory after executing the sequence of actions given by the trace. We first define the evaluation of a single action φ by $[\![\varphi]\!]_{\mathsf{lib}} : \Sigma \to (2^{\Sigma} \cup \{\top\})$:

$$[\![(t, c)]\!]_{\mathsf{lib}}\theta = f_c^t(\theta); \qquad [\![(t, \mathsf{call}\ m(\theta_0))]\!]_{\mathsf{lib}}\theta = \text{if } (\theta * \theta_0)\!\downarrow \text{ then } \{\theta * \theta_0\} \text{ else } \emptyset;$$
$$[\![(t, \mathsf{ret}\ m(\theta_0))]\!]_{\mathsf{lib}}\theta = \text{if } (\theta \setminus \theta_0)\!\downarrow \text{ then } \{\theta \setminus \theta_0\} \text{ else } \top.$$

The evaluation of call and return actions follows their ownership transfer reading explained in Section 3: upon a call to a library, the latter gets the ownership of the specified piece of state; upon a return, the library gives it up. In the former case, only transfers of states compatible with the current library state are allowed. In the latter case, the computation faults when the required piece of state is not available, which ensures that the library respects the contract with its client.

Let us lift $[\![\varphi]\!]_{\mathsf{lib}}$ to 2^{Σ} pointwise: for $p \in 2^{\Sigma}$ we let $[\![\varphi]\!]_{\mathsf{lib}}p = \bigcup\{[\![\varphi]\!]_{\mathsf{lib}}\theta \mid \theta \in p\}$, if $\forall\theta \in p.\, [\![\varphi]\!]_{\mathsf{lib}}\theta \neq \top$; otherwise, $[\![\varphi]\!]_{\mathsf{lib}}p = \top$. We then define the evaluation $[\![\xi]\!]_{\mathsf{lib}} : 2^{\Sigma} \to (2^{\Sigma} \cup \{\top\})$ of a library trace ξ as follows:

$$[\![\varepsilon]\!]_{\mathsf{lib}}p = p; \quad [\![\xi\varphi]\!]_{\mathsf{lib}}p = \text{if } ([\![\xi]\!]_{\mathsf{lib}}p \neq \top) \text{ then } [\![\varphi]\!]_{\mathsf{lib}}([\![\xi]\!]_{\mathsf{lib}}p) \text{ else } \top.$$

In the following, we write $[\![\xi]\!]_{\mathsf{lib}}\theta$ for $[\![\xi]\!]_{\mathsf{lib}}(\{\theta\})$. Using trace evaluation, we can define when a particular trace can be safely executed.

Definition 10. *A library trace ξ is **executable** from θ when $[\![\xi]\!]_{\mathsf{lib}}\theta \notin \{\emptyset, \top\}$.*

Proposition 11. *If ξ is a library trace executable from θ, then $\mathsf{history}(\xi)$ is balanced from $\delta(\theta)$.*

Definition 12. *Library traces ξ and ξ' are **equivalent**, written $\xi \sim \xi'$, if $\xi|_t = \xi'|_t$ for all $t \in \mathsf{ThreadID}$, and the projections of ξ and ξ' to non-interface actions are identical.*

Lemma 13 (Rearrangement). *Assume $(\delta(\theta), H) \sqsubseteq (\delta(\theta'), H')$. If a trace ξ' is executable from θ' and $\mathsf{history}(\xi') = H'$, then there exists a trace ξ executable from θ' such that $\mathsf{history}(\xi) = H$ and $\xi \sim \xi'$.*

The proof transforms ξ' into ξ by repeatedly swapping adjacent actions according to a certain strategy to make the history of the trace equal to H. The most subtle place in the proof is swapping $(t_1, \mathsf{ret}\ m_1(\theta_1))$ and $(t_2, \mathsf{call}\ m_2(\theta_2))$, where $t_1 \neq t_2$. The justification of this transformation relies on the fact that the target history H is balanced. Consider the case when $\theta_1 = \theta_2 = \theta$. Then the two actions correspond to the library first transferring θ to the client and then getting it back. It is impossible for the client to transfer θ to the library earlier, unless it already owned θ before the return in the original trace (this may happen when θ describes only partial permissions for a piece of memory, and thus, its instances can be owned by the client and the library at the

same time). Fortunately, using the fact that H is balanced, we can prove that the latter is indeed the case, and hence, the actions commute.

So far we have used the notion of linearizability on histories, without taking into account library implementations that generate them. In the rest of the paper, we lift this notion to libraries, written in a particular programming language, and prove an Abstraction Theorem, which guarantees that a library can be replaced by another library linearizing it when we reason about its client program.

5 Programming Language

We consider a simple concurrent programming language:

$$C ::= c \mid m \mid C; C \mid C + C \mid C^* \quad L ::= \{m{=}C; \ \ldots; \ m{=}C\} \quad \mathcal{S} ::= \mathsf{let}\, L\, \mathsf{in}\, C \parallel \ldots \parallel C$$

A program consists of a ***library*** L implementing methods $m \in \mathsf{Method}$ and its ***client*** $C_1 \parallel \ldots \parallel C_n$, given by a parallel composition of threads. The commands include primitive commands $c \in \mathsf{PComm}$, method calls $m \in \mathsf{Method}$, sequential composition $C; C'$, nondeterministic choice $C + C'$ and iteration C^*. We use $+$ and * instead of conditionals and while loops for theoretical simplicity: the latter can be defined in the language as syntactic sugar. Methods do not take arguments and do not return values: these can be passed via special locations on the heap associated with the identifier of the thread calling the method. We assume that every method called in the program is defined by the library, and that there are no nested method calls.

An ***open program*** is one without a library (denoted $\mathcal{C}$) or a client (denoted $\mathcal{L}$):

$$\mathcal{C} \ ::= \ \mathsf{let}\, [-]\, \mathsf{in}\, C \parallel \ldots \parallel C \qquad \mathcal{L} \ ::= \ \mathsf{let}\, L\, \mathsf{in}\, [-] \qquad \mathcal{P} ::= \mathcal{S} \mid \mathcal{C} \mid \mathcal{L}$$

In $\mathcal{C}$, we allow the client to call methods that are not defined in the program (but belong to the missing library). We call $\mathcal{S}$ a ***complete program***. Open programs represent a library or a client considered in isolation. The novelty of the kind of open programs we consider here is that we allow them to communicate with their environment via ownership transfers. We now define a way to specify a contract this communication follows.

A ***predicate*** is a set of states from Σ, and a ***parameterised predicate*** is a mapping from thread identifiers to predicates. We use the same symbols p, q, r for ordinary and parameterised predicates. When p is a parameterised predicate, we write p_t for the predicate obtained by applying p to a thread t. Both kinds of predicates can be described syntactically, e.g., using separation logic assertions ([14] and Appendix C).

We describe possible ownership transfers between components with the aid of ***method specifications*** Γ, which are sets of Hoare triples $\{p\}\, m\, \{q\}$, at most one for each method. Here p and q are parameterised predicates such that p_t describes pieces of state transferred when thread t calls the method m, and q_t, those transferred at its return. Note that the pre- and postconditions in method specifications only identify the areas of memory transferred; in other words, they describe the "type" of the returned data structure, but not its "value". As usual for concurrent algorithms, a complete specification of a library is given by its abstract implementation (Section 7).

For example, as we discussed in Section 1, clients of a memory allocator transfer the ownership of memory cells at calls to and returns from it. In particular, the specifications of the allocator methods look approximately as follows:

$$\{\mathsf{emp}\}\mathtt{alloc}\{(\mathtt{r} = 0 \wedge \mathsf{emp}) \vee (\mathtt{r} \neq 0 \wedge \mathsf{Block}(\mathtt{r}))\} \quad \{\mathsf{Block}(\mathtt{blk})\}\mathtt{free}(\mathtt{blk})\{\mathsf{emp}\}$$

Here r denotes the return value of alloc; blk, the actual parameter of free; emp, the empty heap ϵ; and $\mathsf{Block}(\mathtt{r})$, a block of memory at address r managed by the allocator.

To define the semantics of ownership transfers unambiguously, we require pre- and postconditions to be *precise*.

Definition 14. *A predicate $r \in 2^{\Sigma}$ is **precise** [13] if for every state θ there exists at most one substate θ_1 satisfying r, i.e., such that $\theta_1 \in r$ and $\theta = \theta_1 * \theta_2$ for some θ_2.*

Note that, since the $*$ operation is cancellative, when such a substate θ_1 exists, the corresponding substate θ_2 is unique and is denoted by $\theta \setminus r$. Informally, a precise predicate carves out a unique piece of the heap. A parameterised predicate r is precise if so is r_t for every t.

A ***specified open program*** is of the form $\Gamma \vdash \mathcal{C}$ or $\mathcal{L} : \Gamma$. In the former, the specification Γ describes all the methods without implementations that $\mathcal{C}$ may call. In the latter, Γ provides specifications for the methods in the open program that can be called by its external environment. In both cases, Γ specifies the type of another open program that can fill in the hole in $\mathcal{C}$ or $\mathcal{L}$. When we are not sure which form a program has, we write $\Gamma \vdash \mathcal{P} : \Gamma'$, where Γ is empty if $\mathcal{P}$ does not have a client, Γ' is empty if it does not have a library, and both of them are empty if the program is complete.

For open programs $\Gamma \vdash \mathcal{C} = \mathsf{let}\ [-]\ \mathsf{in}\ C_1 \parallel \ldots \parallel C_n$ and $\mathcal{L} : \Gamma = \mathsf{let}\ L\ \mathsf{in}\ [-]$, we denote by $\mathcal{C}(\mathcal{L})$ the complete program $\mathsf{let}\ L\ \mathsf{in}\ C_1 \parallel \ldots \parallel C_n$.

6 Client-Local and Library-Local Semantics

We now give the semantics to complete and open programs. In the latter case, we define component-local semantics that include all behaviours of an open program under any environment satisfying the specification associated with it. In Section 7, we use these to lift linearizability to libraries and formulate the Abstraction Theorem.

We define program semantics in two stages. First, given a program, we generate the set of all its traces possible. This is done solely based on the structure of its statements, without taking into account restrictions arising from the semantics of primitive commands or ownership transfers. The next step filters out traces that are not consistent with these restrictions using a trace evaluation process similar to that in Section 4.

Trace sets. Consider a program $\Gamma \vdash \mathcal{P} : \Gamma'$ and let $M \subseteq \mathsf{Method}$ be the set of methods implemented by its library or called by its client. We define the trace set $(\!|\Gamma \vdash \mathcal{P} : \Gamma'|\!) \in 2^{\mathsf{Trace}}$ of $\mathcal{P}$ in Figure 1. We first define the trace set $(\!|C|\!)^{\Gamma}_{t} S$ of a command C, parameterised by the identifier t of the thread executing it, a method specification Γ, and a mapping $S \in M \times \mathsf{ThreadID} \to 2^{\mathsf{Trace}}$ giving the trace set of the body of every method that C can call when executed by a given thread. The trace set of a client $(\!|C_1 \parallel \ldots \parallel C_n|\!)^{\Gamma} S$ is obtained by interleaving traces of its threads.

The trace set $(\!|\mathcal{C}(\mathcal{L})|\!)$ of a complete program is that of its client computed with respect to a mapping $\lambda m, t.\, (\!|C_m|\!)_{\bar{t}}(\_)$ associating every method m with the trace set of its body C_m. Since we prohibit nested method calls, $(\!|C_m|\!)$ does not depend on the Γ and S parameters. Since the program is complete, we use a method specification Γ_ϵ with empty pre- and postconditions for computing $(\!|C_1 \parallel \ldots \parallel C_n|\!)$. We prefix-close the resulting trace set to take into account incomplete executions. A program $\Gamma \vdash \mathcal{C}$ generates client traces $(\!|\Gamma \vdash \mathcal{C}|\!)$, which do not include internal library actions. This is enforced by associating an empty trace with every library method. Finally, a program $\mathcal{L} : \Gamma'$ generates all possible library traces $(\!|\mathcal{L} : \Gamma'|\!)$. This is achieved by running

$$(\!|c|\!)^{\Gamma}_t S = \{(t, c)\}; \qquad (\!|C_1 + C_2|\!)^{\Gamma}_t S = (\!|C_1|\!)^{\Gamma}_t S \cup (\!|C_2|\!)^{\Gamma}_t S; \qquad (\!|C^*|\!)^{\Gamma}_t S = (((\!|C|\!)^{\Gamma}_t)S)^*;$$
$$(\!|m|\!)^{\Gamma}_t S = \{(t, \mathsf{call}\ m(\theta_p))\,\tau\,(t, \mathsf{ret}\ m(\theta_q)) \mid \tau \in S(m,t) \wedge \theta_p \in p^m_t \wedge \theta_q \in q^m_t\};$$
$$(\!|C_1; C_2|\!)^{\Gamma}_t S = \{\tau_1\tau_2 \mid \tau_1 \in (\!|C_1|\!)^{\Gamma}_t S \wedge \tau_2 \in (\!|C_2|\!)^{\Gamma}_t S\};$$
$$(\!|C_1 \parallel \ldots \parallel C_n|\!)^{\Gamma} S = \bigcup\{\tau_1 \parallel \ldots \parallel \tau_n \mid \forall t = 1..n.\, \tau_t \in (\!|C_t|\!)^{\Gamma}_t S\};$$
$$(\!|\mathsf{let}\ \{m = C_m \mid m \in M\}\ \mathsf{in}\ C_1 \parallel \ldots \parallel C_n|\!) = \mathsf{prefix}((\!|C_1 \parallel \ldots \parallel C_n|\!)^{\Gamma_\epsilon}(\lambda m, t.\, (\!|C_m|\!)^{\sim}_t(\_)));$$
$$(\!|\Gamma \vdash \mathsf{let}\ [-]\ \mathsf{in}\ C_1 \parallel \ldots \parallel C_n|\!) = \mathsf{prefix}((\!|C_1 \parallel \ldots \parallel C_n|\!)^{\Gamma}(\lambda m, t.\, \{\varepsilon\}));$$
$$(\!|\mathsf{let}\ \{m = C_m \mid m \in M\}\ \mathsf{in}\ [-] : \Gamma|\!) = \mathsf{prefix}\left(\textstyle\bigcup_{k\geq 1} (\!|C_{\mathsf{mgc}} \parallel \ldots (k\ \text{times}) \ldots \parallel C_{\mathsf{mgc}}|\!)^{\Gamma}(\lambda m, t.\, (\!|C_m|\!)^{\sim}_t(\_))\right).$$

Fig. 1. Trace sets of commands and programs. Here $\Gamma_\epsilon = \{\{\{\epsilon\}\}\, m\, \{\{\epsilon\}\}\} \mid m \in M\}$, $\Gamma = \{\{p^m\}\, m\, \{q^m\} \mid m \in M\}$, $M = \{m_1, \ldots, m_j\}$, $C_{\mathsf{mgc}} = (m_1 + \ldots + m_j)^*$, and $\mathsf{prefix}(T)$ is the prefix closure of T. Also, $\tau \in \tau_1 \parallel \ldots \parallel \tau_n$ if and only if every action in τ is done by a thread $t \in \{1, \ldots, n\}$ and for all such t, we have $\tau|_t = \tau_t$.

the library under its *most general client*, where every thread executes an infinite loop, repeatedly invoking arbitrary library methods.

Evaluation. The set of traces generated using $(\!|\cdot|\!)$ may include those not consistent with the semantics of primitive commands or expected ownership transfers. We therefore define the meaning of a program $[\![\Gamma \vdash \mathcal{P} : \Gamma']\!] \in \Sigma \to (2^{\mathsf{Trace}} \cup \{\top\})$ by evaluating every trace in $(\!|\Gamma \vdash \mathcal{P} : \Gamma'|\!)$ to determine whether it is executable.

First, consider a library $\mathcal{L} : \Gamma'$. In this case we use the evaluation function $[\![\cdot]\!]_{\mathsf{lib}}$ defined in Section 4. We let $[\![\mathcal{L} : \Gamma']\!]\theta = \top$, if

$$\begin{aligned}\exists \xi, t.\ &(\exists c.\ [\![\xi]\!]_{\mathsf{lib}}\theta \neq \top \wedge \xi\,(t, c) \in (\!|\mathcal{L} : \Gamma'|\!) \wedge f^t_c(\theta) = \top) \vee \\ &(\exists m, \theta_q.\ [\![\xi]\!]_{\mathsf{lib}}\theta \neq \top \wedge \xi\,(t, \mathsf{ret}\ m(\theta_q)) \in (\!|\mathcal{L} : \Gamma'|\!) \\ &\qquad \wedge \forall \theta'_q.\ \xi\,(t, \mathsf{ret}\ m(\theta'_q)) \in (\!|\mathcal{L} : \Gamma'|\!) \Rightarrow [\![\xi\,(t, \mathsf{ret}\ m(\theta'_q))]\!]_{\mathsf{lib}}\theta = \top).\end{aligned}$$

Thus, the library has no semantics if a primitive command in one of its executions faults, or the required piece of state is not available for transferring to the client at a method return. Otherwise, $[\![\mathcal{L} : \Gamma']\!]\theta = \{\xi \mid \xi \in (\!|\mathcal{L} : \Gamma'|\!) \wedge [\![\xi]\!]_{\mathsf{lib}}\theta \notin \{\emptyset, \top\}\}$. This gives a ***library-local*** semantics to $\mathcal{L}$, in the sense that it takes into account only the part of the program state owned by the library and considers its behaviour under any client respecting Γ'. This generalises the standard notion of the most general client to situations where the library performs ownership transfers. Lemma 16 below confirms that the client defined by $(\!|\mathcal{L} : \Gamma'|\!)$ and $[\![\cdot]\!]_{\mathsf{lib}}$ is indeed most general, as it reproduces library behaviours under any possible clients.

To give a semantics to $\Gamma \vdash \mathcal{C}$, we define an evaluation function $[\![\eta]\!]_{\mathsf{client}} : 2^{\Sigma} \to (2^{\Sigma} \cup \{\top\})$ for client traces η. To this end, we define the evaluation of a single action φ by $[\![\varphi]\!]_{\mathsf{client}} : \Sigma \to (2^{\Sigma} \cup \{\top\})$ and then lift it to client traces as in Section 4:

$$[\![(t, c)]\!]_{\mathsf{client}}\theta = f^t_c(\theta); \qquad [\![(t, \mathsf{call}\ m(\theta_0))]\!]_{\mathsf{client}}\theta = \text{if } (\theta \setminus \theta_0)\downarrow \text{ then } \{\theta \setminus \theta_0\} \text{ else } \top;$$
$$[\![(t, \mathsf{ret}\ m(\theta_0))]\!]_{\mathsf{client}}\theta = \text{if } (\theta * \theta_0)\downarrow \text{ then } \{\theta * \theta_0\} \text{ else } \emptyset.$$

When a thread t calls a method m in Γ, it transfers the ownership of the specified piece of state to the library being called. The evaluation faults if the state to be transferred is not available, which ensures that the client respects the specifications of the library.

When the method returns, the client receives the ownership of the specified piece of state, which has to be compatible with the state of the client. We let $[\![\Gamma \vdash \mathcal{C}]\!]\theta = \top$, if

$$\exists \eta, t.\ (\exists c.\ [\![\eta]\!]_{\mathsf{client}}\theta \neq \top \wedge \eta\,(t, c) \in (\!|\Gamma \vdash \mathcal{C}|\!) \wedge f_c^t(\theta) = \top) \vee \\ (\exists m, \theta_p.\ [\![\eta]\!]_{\mathsf{client}}\theta \neq \top \wedge \eta\,(t, \mathsf{call}\ m(\theta_p)) \in (\!|\Gamma \vdash \mathcal{C}|\!) \\ \wedge \forall \theta'_p.\ \eta\,(t, \mathsf{call}\ m(\theta'_p)) \in (\!|\Gamma \vdash \mathcal{C}|\!) \Rightarrow [\![\eta\,(t, \mathsf{ret}\ m(\theta'_p))]\!]_{\mathsf{client}}\theta = \top).$$

Otherwise, $[\![\Gamma \vdash \mathcal{C}]\!]\theta = \{\eta \mid \eta \in (\!|\Gamma \vdash \mathcal{C}|\!) \wedge [\![\eta]\!]_{\mathsf{client}}\theta \notin \{\emptyset, \top\}\}$. This gives a ***client-local*** semantics to $\mathcal{C}$, in the sense that it takes into account only the part of the state owned by the client and considers its behaviour when using any library respecting Γ.

Finally, for a complete program $\mathcal{C}(\mathcal{L})$, we let $[\![\mathcal{C}(\mathcal{L})]\!]\theta = \top$, if $\exists \tau.\ \tau \in (\!|\mathcal{C}(\mathcal{L})|\!) \wedge [\![\tau]\!]_{\mathsf{lib}}\theta = \top$; otherwise, $[\![\mathcal{C}(\mathcal{L})]\!]\theta = \{\tau \mid \tau \in (\!|\mathcal{C}(\mathcal{L})|\!) \wedge [\![\tau]\!]_{\mathsf{lib}}\theta \neq \emptyset\}$ (note that using $[\![\cdot]\!]_{\mathsf{client}}$ here would yield the same result). For a set of initial states $I \subseteq \Sigma$, let

$$[\![(\Gamma \vdash \mathcal{P} : \Gamma'), I]\!] = \{(\theta, \tau) \mid \theta \in I \wedge \tau \in [\![\Gamma \vdash \mathcal{P} : \Gamma']\!]\theta\}.$$

Definition 15. *A program $\Gamma \vdash \mathcal{P} : \Gamma'$ is **safe** at θ, if $[\![\Gamma \vdash \mathcal{P} : \Gamma']\!]\theta \neq \top$; $\mathcal{P}$ is safe for $I \subseteq \Sigma$, if it is safe at θ for all $\theta \in I$.*

Commands fault when accessing memory cells that are not present in the state they are run from. Thus, the safety of a program guarantees that it does not touch the part of the heap belonging to its environment. Besides, calls to methods in Γ and returns from methods in Γ' fault when the piece of state they have to transfer is not available. Thus, the safety of the program also ensures that it respects the contract with its environment given by Γ or Γ'.

While decomposing the verification of a closed program into the verification of its components, we rely on the above properties to ensure that we can indeed reason about the components in isolation, without worrying about the interference from their environment. In particular, our definition of linearizability on libraries considers only safe libraries (Section 7).

Soundness and adequacy. The client-local and library-local semantics are sound and adequate with respect to the global semantics of the complete program. These properties are used in the proof of the Abstraction Theorem.

Let ground be a function on traces that replaces the state annotations θ of all interface actions with ϵ. For a trace τ, we define its projection $\mathsf{client}(\tau)$ to actions executed by the client code: we include $\varphi = (t, \_)$ with $\tau = \tau'\varphi\tau''$ into the projection, if (i) φ is an interface action; or (ii) φ is outside an invocation of a method, i.e., it is not the case that $\tau|_t = \tau_1\,(t, \mathsf{call}\ \_)\,\tau_2\varphi\tau_3$, where τ_2 does not contain a $(t, \mathsf{ret}\ \_)$ action. We also use a similar projection $\mathsf{lib}(\tau)$ to library actions.

The following lemma shows that a trace of $\mathcal{C}(\mathcal{L})$ generates two traces in the client-local and library-local semantics with the same history. The lemma thus carries over properties of the local semantics, such as safety, to the global one, and in this sense is the statement of the soundness of the former with respect to the latter.

Lemma 16 (Soundness). *Assume $\Gamma \vdash \mathcal{C}$ and $\mathcal{L} : \Gamma$ safe for I_1 and I_2, respectively. Then so is $\mathcal{C}(\mathcal{L})$ for $I_1 * I_2$ and*

$$\forall (\theta, \tau) \in [\![\mathcal{C}(\mathcal{L}), I_1 * I_2]\!].\ \exists (\theta_1, \eta) \in [\![\mathcal{C}, I_1]\!].\ \exists (\theta_2, \xi) \in [\![\mathcal{L}, I_2]\!].\ \theta = \theta_1 * \theta_2 \wedge \\ \mathsf{history}(\eta) = \mathsf{history}(\xi) \wedge \mathsf{client}(\tau) = \mathsf{ground}(\eta) \wedge \mathsf{lib}(\tau) = \mathsf{ground}(\xi).$$

The following lemma states that any pair of client-local and library-local traces agreeing on the history can be combined into a trace of $\mathcal{C}(\mathcal{L})$. It thus carries over properties of the global semantics to the local ones, stating the adequacy of the latter.

Lemma 17 (Adequacy). *If $\mathcal{L} : \Gamma$ and $\Gamma \vdash \mathcal{C}$ are safe for I_1 and I_2, respectively, then*

$$\forall(\theta_1, \eta) \in [\![\mathcal{C}, I_1]\!]. \forall(\theta_2, \xi) \in [\![\mathcal{L}, I_2]\!]. ((\theta_1 * \theta_2)\downarrow \wedge \mathsf{history}(\eta) = \mathsf{history}(\xi)) \Rightarrow \\ \exists\tau. (\theta_1 * \theta_2, \tau) \in [\![\mathcal{C}(\mathcal{L}), I_1 * I_2]\!] \wedge \mathsf{client}(\tau) = \mathsf{ground}(\eta) \wedge \mathsf{lib}(\tau) = \mathsf{ground}(\xi).$$

7 Abstraction Theorem

We are now in a position to lift the notion of linearizability on histories to libraries and prove the central technical result of this paper—the Abstraction Theorem. We define linearizability between specified libraries $\mathcal{L} : \Gamma$, together with their sets of initial states I. First, using the library-local semantics, we define the set of histories of a library $\mathcal{L}$ with the set of initial states I: $\mathsf{history}(\mathcal{L}, I) = \{(\delta(\theta_0), \mathsf{history}(\tau)) \mid (\theta_0, \tau) \in [\![\mathcal{L}, I]\!]\}$.

Definition 18. *Consider $\mathcal{L}_1 : \Gamma$ and $\mathcal{L}_2 : \Gamma$ safe for I_1 and I_2, respectively. We say that $(\mathcal{L}_2, I_2)$ **linearizes** $(\mathcal{L}_1, I_1)$, written $(\mathcal{L}_1, I_1) \sqsubseteq (\mathcal{L}_2, I_2)$, if*

$$\forall(l_1, H_1) \in \mathsf{history}(\mathcal{L}_1, I_1). \exists(l_2, H_2) \in \mathsf{history}(\mathcal{L}_2, I_2). (l_1, H_1) \sqsubseteq (l_2, H_2).$$

Thus, $(\mathcal{L}_2, I_2)$ linearizes $(\mathcal{L}_1, I_1)$ if every behaviour of the latter may be reproduced in a linearized form by the former without requiring more memory.

Theorem 19 (Abstraction). *If $\mathcal{L}_1 : \Gamma$, $\mathcal{L}_2 : \Gamma$, $\Gamma \vdash \mathcal{C}$ are safe for I_1, I_2, I, respectively, and $(\mathcal{L}_1, I_1) \sqsubseteq (\mathcal{L}_2, I_2)$, then $\mathcal{C}(\mathcal{L}_1)$ and $\mathcal{C}(\mathcal{L}_2)$ are safe for $I * I_1$ and $I * I_2$, respectively, and*

$$\forall(\theta_1, \tau_1) \in [\![\mathcal{C}(\mathcal{L}_1), I * I_1]\!]. \exists(\theta_2, \tau_2) \in [\![\mathcal{C}(\mathcal{L}_2), I * I_2]\!]. \mathsf{client}(\tau_1) = \mathsf{client}(\tau_2).$$

Thus, when reasoning about a client $\mathcal{C}(\mathcal{L}_1)$ of a library $\mathcal{L}_1$, we can soundly replace $\mathcal{L}_1$ with a library $\mathcal{L}_2$ linearizing it: if a linear-time safety property over client actions holds of $\mathcal{C}(\mathcal{L}_2)$, it will also hold of $\mathcal{C}(\mathcal{L}_1)$. In practice, we are usually interested in ***atomicity abstraction***, a special case of this transformation when methods in $\mathcal{L}_2$ are atomic. The theorem is restricted to safety properties as, for simplicity, in this paper we consider only finite histories and traces. Our results can be generalised to the infinite case as in [10]. The requirement that $\mathcal{C}$ be safe in the theorem restricts its applicability to *healthy* clients that do not access library internals.

To prove Theorem 19, we first lift Lemma 13 to traces in the library-local semantics.

Corollary 20. *If $(\delta(\theta), H) \sqsubseteq (\delta(\theta'), H')$ and $\mathcal{L}$ is safe at θ', then*

$$\forall\xi' \in [\![\mathcal{L}]\!]\theta'. \mathsf{history}(\xi') = H' \Rightarrow \exists\xi \in [\![\mathcal{L}]\!]\theta'. \mathsf{history}(\xi) = H.$$

Thus, if $(\mathcal{L}_1, I_1) \sqsubseteq (\mathcal{L}_2, I_2)$, then the set of histories of $\mathcal{L}_1$ is a subset of those of $\mathcal{L}_2$: linearizability is a sound criterion for proving that one library simulates another.

Proof of Theorem 19. The safety of $\mathcal{C}(\mathcal{L}_1)$ and $\mathcal{C}(\mathcal{L}_2)$ follows from Lemma 16. Take $(\theta, \tau_1) \in [\![\mathcal{C}(\mathcal{L}_1), I * I_1]\!]$. We transform the trace τ_1 of $\mathcal{C}(\mathcal{L}_1)$ into a trace τ_2 of $\mathcal{C}(\mathcal{L}_2)$ with the same client projection using the local semantics of $\mathcal{L}_1$, $\mathcal{L}_2$ and $\mathcal{C}$. Namely, we first apply Lemma 16 to generate a pair of a library-local initial state and a trace

$(\theta_l^1, \xi_1) \in [\![\mathcal{L}_1, I_1]\!]$ and a client-local pair $(\theta_c, \eta) \in [\![\mathcal{C}, I]\!]$, such that $\theta = \theta_c * \theta_l^1$, $\mathsf{client}(\tau_1) = \mathsf{ground}(\eta)$ and $\mathsf{history}(\eta) = \mathsf{history}(\xi_1)$. Since $(\mathcal{L}_1, I_1) \sqsubseteq (\mathcal{L}_2, I_2)$, for some $(\theta_l^2, \xi_2) \in [\![\mathcal{L}_2, I_2]\!]$, we have $\delta(\theta_l^2) \preceq \delta(\theta_l^1)$ and $(\delta(\theta_l^1), \mathsf{history}(\xi_1)) \sqsubseteq (\delta(\theta_l^2), \mathsf{history}(\xi_2))$. By Corollary 20, ξ_2 can be transformed into a trace ξ_2' such that $(\theta_l^2, \xi_2') \in [\![\mathcal{L}_2, I_2]\!]$ and $\mathsf{history}(\xi_2') = \mathsf{history}(\xi_1) = \mathsf{history}(\eta)$. Since $\delta(\theta_l^2) \preceq \delta(\theta_l^1)$ and $(\theta_c * \theta_l^1)\!\downarrow$, we have $(\theta_c * \theta_l^2)\!\downarrow$. We then use Lemma 17 to compose the library-local trace ξ_2' with the client-local one η into a trace τ_2 such that $(\theta_c * \theta_l^2, \tau_2) \in [\![\mathcal{C}(\mathcal{L}_2), I * I_2]\!]$ and $\mathsf{client}(\tau_2) = \mathsf{ground}(\eta) = \mathsf{client}(\tau_1)$. □

Establishing linearizability with ownership transfer and its applications. We have developed a logic for proving linearizability in the sense of Definition 18, which generalises an existing proof system [16] based on separation logic [14] to the setting with ownership transfer. The logic uses the usual method of proving linearizability based on linearization points [1, 12, 16] and treats ownership transfers between a library and its environment in the same way as transfers between procedures and their callers in separation logic. Due to space constraints, the details of the logic are beyond the scope of this paper and are described in Appendix D. We mention the logic here to emphasise that our notion of linearizability can indeed be established effectively.

The Abstraction Theorem is not just a theoretical result: it enables compositional reasoning about complex concurrent algorithms that are challenging for existing verification methods. For example, the theorem can be used to justify Vafeiadis's compositional proof [16, Section 5.3] of the multiple-word compare-and-swap (MCAS) algorithm implemented using an auxiliary operation called RDCSS [11] (the proof used an abstraction of the kind enabled by Theorem 19 without justifying its correctness). If the MCAS algorithm were verified together with RDCSS, its proof would be extremely compicated. Fortunately, we can consider MCAS as a client of RDCSS, with the two components performing ownership transfers between them. The Abstraction Theorem then makes the proof tractable by allowing us to verify the linearizability of MCAS assuming an atomic specification of the inner RDCSS algorithm.

8 Frame Rule for Linearizability

Libraries such as concurrent containers are used by clients to transfer the ownership of data structures, but do not actually access their contents. We show that for such libraries, the classical linearizability implies linearizability with ownership transfer.

Definition 21. *A method specification* $\Gamma' = \{\{r^m\}\, m\, \{s^m\} \mid m \in M\}$ ***extends*** *a specification* $\Gamma = \{\{p^m\}\, m\, \{q^m\} \mid m \in M\}$*, if* $\forall t.\, r_t^m \subseteq p_t^m * \Sigma \wedge s_t^m \subseteq q_t^m * \Sigma$.

For example, Γ might say that a method m receives a pointer x as a parameter: $\{\exists x.\, \mathtt{param}[t] \mapsto x\}\, m\, \{\mathtt{param}[t] \mapsto \_\}$, where t is the identifier of the thread calling m. Then Γ' may mandate that the cell the pointer identifies be transferred to the method: $\{\exists x.\, \mathtt{param}[t] \mapsto x * x \mapsto \_\}\, m\, \{\mathtt{param}[t] \mapsto \_\}$. For a history H, let $[\![H]\!]_\Gamma$ be the result of replacing every action φ in H by the action $[\![\varphi]\!]_\Gamma$ defined as follows:

$$[\![(t, \mathsf{call}\ m(\theta))]\!]_\Gamma = (t, \mathsf{call}\ m(\theta \setminus p_t^m)); \quad [\![(t, \mathsf{ret}\ m(\theta))]\!]_\Gamma = (t, \mathsf{ret}\ m(\theta \setminus q_t^m)).$$

$[\![H]\!]_\Gamma$ is undefined if so is the result of any of the $\setminus$ operations above. The operation selects the extra pieces of state not required by Γ.

Theorem 22 (Frame rule). *Assume (i) for all $i \in \{1,2\}$, $\mathcal{L}_i : \Gamma$ and $\mathcal{L}_i : \Gamma'$ are safe for I_i and $I_i * I$, respectively; (ii) $(\mathcal{L}_1 : \Gamma, I_1) \sqsubseteq (\mathcal{L}_2 : \Gamma, I_2)$; (iii) Γ' extends Γ; and (iv) for every $(\theta_0, \theta_0') \in I_1 \times I$ and $\xi \in [\![\mathcal{L}_1 : \Gamma']\!](\theta_0 * \theta_0')$, the trace $\lfloor\!\lfloor \mathsf{history}(\xi) \rfloor\!\rfloor_\Gamma$ is executable from θ_0'. Then $(\mathcal{L}_1 : \Gamma', I_1 * I) \sqsubseteq (\mathcal{L}_2 : \Gamma', I_2 * I)$.*

The proof of the theorem relies on Corollary 20. The linearizability relation established in the theorem enables the use of the Abstraction Theorem for clients performing ownership transfer. The safety requirement on $\mathcal{L}_1$ and $\mathcal{L}_2$ with respect to Γ' is needed because Γ' not only transfers extra memory to the library in its preconditions, but also takes it back in its postconditions. The requirement (iv) ensures that the extra memory required by postconditions in Γ' comes from the extra memory provided in its preconditions and the extension of the initial state, not from the memory transferred according to Γ.

9 Related Work

In our previous work, we proved Abstraction Theorems for definitions of linearizability supporting reasoning about liveness properties [10] and weak memory models [4]. These definitions assumed that the library and its client operate in disjoint address spaces and, hence, are guaranteed not to interfere with each other and cannot communicate via the heap. Lifting this restriction is the goal of the present paper. Although we borrow the basic proof structure of Theorem 19 from [4], including the split into Lemmas 13, 16 and 17, the formulations and proofs of the Abstraction Theorem and the lemmas here have to deal with technical challenges posed by ownership transfer that did not arise in previous work. First, their formulations rely on the novel forms of client-local and library-local semantics, and in particular, the notion of the most general client (Section 6), that allow a component to communicate with its environment via ownership transfers. Proving Lemmas 16 and 17 then involves a delicate tracking of a splitting between the parts of the state owned by the library and the client, and how ownership transfers affect it. Second, the key result needed to establish the Abstraction Theorem is the Rearrangement Lemma (Lemma 13). What makes the proof of this lemma difficult in our case is the need to deal with subtle interactions between concurrency and ownership transfer that have not been considered in previous work. Namely, changing the history of a sequential library specification for one of its concurrent implementation in the lemma requires commuting ownership transfer actions; justifying the correctness of these transformations is non-trivial and relies on the notion of history balancedness that we propose.

Recently, there has been a lot of work on verifying linearizability of common algorithms; representative papers include [1,7,16]. All of them proved classical linearizability, where libraries and their clients exchange values of a given data type and do not perform ownership transfers. This includes even libraries such as concurrent containers discussed in Section 1, that are actually used by client threads to transfer the ownership of data structures. The frame rule for linearizability we propose (Theorem 22) justifies that classical linearizability established for concurrent containers entails linearizability with ownership transfer. This makes our Abstraction Theorem applicable, enabling compositional reasoning about their clients.

Turon and Wand [15] have proposed a logic for establishing refinements between concurrent modules, likely equivalent to linearizability. Their logic considers libraries and clients residing in a shared address space, but not ownership transfer. It assumes that the client does not access the internal library state; however, their paper does not

provide a way of checking this condition. As a consequence, Turon and Wand do not propose an Abstraction Theorem strong enough to support separate reasoning about a library and its client in realistic situations of the kind we consider.

Elmas et al. [7,8] have developed a system for verifying concurrent programs based on repeated applications of atomicity abstraction. They do not use linearizability to perform the abstraction. Instead, they check the commutativity of an action to be incorporated into an atomic block with *all* actions of other threads. In particular, to abstract a library implementation in a program by its atomic specification, their method would have to check the commutativity of every internal action of the library with all actions executed by the client code of other threads. Thus, the method of Elmas et al. does not allow decomposing the verification of a program into verifying libraries and their clients separately. In contrast, our Abstraction Theorem ensures the atomicity of a library under *any* healthy client.

Ways of establishing relationships between different sequential implementations of the same library have been studied in *data refinement*, including cases of interactions via ownership transfer [2,9]. Our results can be viewed as generalising data refinement to the concurrent setting.

Acknowledgements. We would like to thank Anindya Banerjee, Josh Berdine, Xinyu Feng, Hongjin Liang, David Naumann, Peter O'Hearn, Matthew Parkinson, Noam Rinetzky and Julles Villard for helpful comments. Yang was supported by EPSRC.

References

1. D. Amit, N. Rinetzky, T. W. Reps, M. Sagiv, and E. Yahav. Comparison under abstraction for verifying linearizability. In *CAV*, 2007.
2. A. Banerjee and D. A. Naumann. Ownership confinement ensures representation independence in object-oriented programs. *JACM*, 52(6), 2005.
3. R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permission accounting in separation logic. In *POPL*, 2005.
4. S. Burckhardt, A. Gotsman, M. Musuvathi, and H. Yang. Concurrent library correctness on the TSO memory model. In *ESOP*, 2012.
5. C. Calcagno, P. O'Hearn, and H. Yang. Local action and abstract separation logic. In *LICS*, 2007.
6. D. G. Clarke, J. Noble, and J. M. Potter. Simple ownership types for object containment. In *ECOOP*, 2001.
7. T. Elmas, S. Qadeer, A. Sezgin, O. Subasi, and S. Tasiran. Simplifying linearizability proofs with reduction and abstraction. In *TACAS*, 2010.
8. T. Elmas, S. Qadeer, and S. Tasiran. A calculus of atomic actions. In *POPL*, 2009.
9. I. Filipović, P. O'Hearn, N. Torp-Smith, and H. Yang. Blaiming the client: On data refinement in the presence of pointers. *FAC*, 22(5), 2010.
10. A. Gotsman and H. Yang. Liveness-preserving atomicity abstraction. In *ICALP*, 2011.
11. T. Harris, K. Fraser, and I. Pratt. A practical multi-word compare-and-swap operation. In *DISC*, 2002.
12. M. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *TOPLAS*, 12(3), 1990.
13. P. O'Hearn. Resources, concurrency and local reasoning. *TCS*, 375(1-3), 2007.
14. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, 2002.
15. A. Turon and M. Wand. A separation logic for refining concurrent objects. In *POPL*, 2011.
16. V. Vafeiadis. Modular fine-grained concurrency verification. PhD Thesis. University of Cambridge, 2008.
17. V. Vafeiadis. Automatically proving linearizability. In *CAV*, 2010.

A Additional Definitions

Conditions on the transformers for primitive commands. We place the following restrictions on f_c^t for every primitive command $c \in \mathsf{PComm}$ and thread $t \in \mathsf{ThreadID}$:

- ***Footprint Preservation:*** for all $\theta, \theta' \in \Sigma$, if $\theta' \in f_c^t(\theta)$, then $\delta(\theta') = \delta(\theta)$.
- ***Strong Locality:*** for all θ_1, θ_2, if $(\theta_1 * \theta_2)\!\downarrow$ and $f_c^t(\theta_1) \neq \top$, then $f_c^t(\theta_1 * \theta_2) = f_c^t(\theta_1) * \{\theta_2\}$.

Footprint preservation prohibits primitive commands from allocating or deallocating memory. This does not pose a problem, since in the context of linearizability, an allocator is just another library and should be treated as such. The strong locality of f_c^t says that, if a command c can be safely executed from a state θ_1, then when executed from a bigger state $\theta_1 * \theta_2$, it does not change the additional state θ_2 and its effect depends only on the state θ_1 and not on the additional state θ_2. The strong locality is a strengthening of the locality property in separation logic [5]. While the usual locality prohibits the command from changing the additional state, it permits the effect of the command to depend on this state [9]. On the other hand, such dependency is forbiden by the strong locality.

The transformers for standard commands, except memory (de)allocation, satisfy the above conditions.

In the following, we sometimes use the pointwise lifting $f_c^t : 2^\Sigma \to 2^\Sigma \cup \{\top\}$ of the transformers to sets of states:

$$f_c^t(p) \;=\; \begin{cases} \top, & \text{if } \exists \theta \in p.\ f_c^t(\theta) = \top; \\ \bigcup_{\theta \in p} f_c^t(\theta), & \text{otherwise.} \end{cases}$$

Semantics of typical primitive commands. When our state model Σ is RAM, we typically consider following primitive commands:

$$\mathsf{skip}, \qquad [E] = E', \qquad \mathsf{assume}(E),$$

where expressions E are defined as follows:

$$E \;::=\; \mathbb{Z} \mid \mathsf{tid} \mid [E] \mid E + E \mid -E \mid !E \mid \ldots$$

Here tid refers to the identifier of the thread executing the command, $[E]$ returns the contents of the address E in memory, and $!E$ is the C-style negation of an expression E—it returns 1 when E evaluates to 0, and 0 otherwise. We denote by $[\![E]\!]_{\theta,t} \in \mathsf{Val} \cup \{\top\}$ the result of evaluating the expression E in the state θ with the current thread identifier t. When this evaluation dereferences illegal memory addresses, it results in the error value $\top$.

For the above commands and $t \in \mathsf{ThreadID}$, we define corresponding transition relation $\rightsquigarrow_t$: $\mathsf{RAM} \times (\mathsf{RAM} \cup \{\top\})$ in Figure 2. Using this transition relation, we then define $f_c^t : \mathsf{RAM} \to 2^{\mathsf{RAM}} \cup \{\top\}$ for the primitive commands c as follows:

$$f_c^t(\theta) \;=\; \begin{cases} \top, & \text{if } \theta, c \rightsquigarrow_t \top; \\ \bigcup \{\theta' \mid \theta, c \rightsquigarrow_t \theta'\}, & \text{otherwise.} \end{cases}$$

Example of a separation algebra with permissions. We now present an extension RAM_p of the separation algebra RAM in Section 2, where states carry additional information regarding permissions to access memory cells. Here we consider simple permissions to read from and write to memory cells, often used to model read sharing among

multiple threads or program components.

Formally, the example algebra RAM_p is defined as follows:

$$\begin{array}{l}\mathsf{Loc} = \{1, 2, \ldots\}; \qquad \mathsf{Val} = \mathbb{Z}; \qquad \mathsf{Perm} = (0, 1]; \\ \mathsf{RAM}_p = \mathsf{Loc} \rightharpoonup_{fin} (\mathsf{Val} \times \mathsf{Perm}).\end{array}$$

A state in this model consists of a finite partial function from allocated memory locations to values they store and so-called *permissions*—numbers from $(0, 1]$ that show "how much" of the memory cell belongs to the partial state [3]. The latter allow a library and its client to share access to some of memory cells. Permissions in RAM_p allow only read sharing: when defining the semantics of commands over states in RAM_p, the permissions strictly less than 1 are interpreted as permissions to read; the full permission 1 additionally allows writing. This can be generalised to sharing permissions to access memory in an arbitrary way consistent with a given specification[1].

We remind the reader that for a partial function g, $g(x)\!\downarrow$ means that the function g is defined on x. We also write $g(x)\!\uparrow$ when it is undefined on x.

The $*$ operation on RAM_p adds up permissions for memory cells. Formally, for $\theta_1, \theta_2 \in \mathsf{RAM}_p$, we write $\theta_1 \sharp \theta_2$ when:

$$\begin{array}{l}\forall x \in \mathsf{Loc}.\, \theta_1(x)\!\downarrow \wedge\, \theta_2(x)\!\downarrow \Rightarrow \\ \qquad\qquad (\exists u, \pi_1, \pi_2.\, \theta_1(x) = (u, \pi_1) \;\wedge\; \theta_2(x) = (u, \pi_2) \;\wedge\; \pi_1 + \pi_2 \leq 1).\end{array}$$

If $\theta_1 \sharp \theta_2$, then we define

$$\begin{array}{l}\theta_1 * \theta_2 = \{(x, (u, \pi)) \mid (\theta_1(x) = (u, \pi) \wedge \theta_2(x)\!\uparrow) \vee (\theta_2(x) = (u, \pi) \wedge \theta_1(x)\!\uparrow) \vee \\ \qquad\qquad (\theta_1(x) = (u, \pi_1) \wedge \theta_2(x) = (u, \pi_2) \wedge \pi = \pi_1 + \pi_2)\}.\end{array}$$

Otherwise, $\theta_1 * \theta_2$ is undefined. The unit for $*$ is the empty heap $[\,]$. This definition of $*$ allows us, e.g., to split a memory area into two disjoint parts. It also allows splitting a cell with a full permission 1 into two parts, carrying read-only permissions $1/2$ and agreeing on the value stored in the cell. These permissions can later be recombined to obtain the full permission, which allows both reading from and writing to the cell.

In the case of the algebra RAM_p, for $\theta \in \mathsf{RAM}_p$ we have

$$\begin{array}{l}\delta(\theta) = \{\theta' \mid \forall x.\, (\theta(x)\!\downarrow \Leftrightarrow \theta'(x)\!\downarrow) \wedge \forall u, \pi.\, (\theta(x) = (u, \pi) \wedge \pi < 1 \Rightarrow \\ \qquad\qquad \theta(x) = \theta'(x)) \wedge (\theta(x) = (u, 1) \Rightarrow \theta'(x) = (\_, 1))\}.\end{array}$$

In other words, states with the same footprint contain the same memory cells with the identical permissions; in the case of memory cells on read permissions, the states also have to agree on their values. It is easy to check that the condition in Definition 3 is satisfied.

Finally, we define $f_c^t : \mathsf{RAM}_p \to 2^{\mathsf{RAM}_p} \cup \{\top\}$ for primitive commands c, following the same recipe as in the RAM case. In this case, we use the transition relation described in Figure 3.

---

[1] M. Dodds, X. Feng, M. Parkinson, and V. Vafeiadis. Deny-guarantee reasoning. In *ESOP*, 2009.

$$
\begin{array}{lll}
\theta, \mathsf{skip} & \rightsquigarrow_t & \theta \\
\theta, [E] = E' & \rightsquigarrow_t & \theta[[\![E]\!]_{\theta,t} : [\![E']\!]_{\theta,t}], \quad \text{if } [\![E]\!]_{\theta,t} \in \mathsf{dom}(\theta), [\![E']\!]_{\theta,t} \in \mathsf{Val} \\
\theta, [E] = E' & \rightsquigarrow_t & \top, \quad \text{if } [\![E]\!]_{\theta,t} \notin \mathsf{dom}(\theta) \text{ or } [\![E']\!]_{\theta,t} = \top \\
\theta, \mathsf{assume}(E) & \rightsquigarrow_t & \theta, \quad \text{if } [\![E]\!]_{\theta,t} \in \mathsf{Val} - \{0\} \\
\theta, \mathsf{assume}(E) & \rightsquigarrow_t & \top, \quad \text{if } [\![E]\!]_{\theta,t} = \top
\end{array}
$$

Fig. 2. Transition relation for sample primitive commands in the RAM model. The result $\top$ indicates that the command faults.

$$
\begin{array}{lll}
\theta, \mathsf{skip} & \rightsquigarrow_t & \theta \\
\theta, [E] = E' & \rightsquigarrow_t & \theta[[\![E]\!]_{\theta,t} : ([\![E']\!]_{\theta,t}, 1)], \quad \text{if } \theta([\![E]\!]_{\theta,t}) = (\_, 1), [\![E']\!]_{\theta,t} \in \mathsf{Val} \\
\theta, [E] = E' & \rightsquigarrow_t & \top, \quad \text{if the above condition does not hold} \\
\theta, \mathsf{assume}(E) & \rightsquigarrow_t & \theta, \quad \text{if } [\![E]\!]_{\theta,t} \in \mathsf{Val} - \{0\} \\
\theta, \mathsf{assume}(E) & \rightsquigarrow_t & \top, \quad \text{if } [\![E]\!]_{\theta,t} = \top
\end{array}
$$

Fig. 3. Transition relation for sample primitive commands in the RAM_p model. The evaluation of expressions $[\![E]\!]_{\theta,t}$ ignores permissions in θ.

B Proofs

B.1 Operational Semantics

It is more convenient for us to do the proofs using an operational semantics of the language of Section 5, rather than its denotational-style semantics given in Section 6.

Control-flow graphs. In the definition of program semantics, it is technically convenient for us to abstract from a particular syntax of programming language and represent commands by their ***control-flow graphs***. A control-flow graph (CFG) is a tuple $(N, T, \mathsf{start}, \mathsf{end})$, consisting of the set of program points N, the control-flow relation $T \subseteq N \times \mathsf{Comm} \times N$, and the initial and final positions $\mathsf{start}, \mathsf{end} \in N$. The edges are annotated with commands from Comm, which are primitive commands or method calls m. Every command C in our language can be translated to a CFG in the standard manner.

We represent a specified program $\Gamma \vdash \mathcal{P} : \Gamma'$ by a collection of CFGs. If $\mathcal{P}$ contains a client with n threads running C_t, $t = 1..n$, each thread t is represented by the CFG $(N_t, T_t, \mathsf{start}_t, \mathsf{end}_t)$ of C_t. Let $\mathsf{Method}(\mathcal{P})$ be the set of all methods declared in $\mathcal{P}$. For each method $m \in \mathsf{Method}(\mathcal{P})$ let C_m be the body of its implementation. Every such method is then represented by the CFG $(N_m, T_m, \mathsf{start}_m, \mathsf{end}_m)$ of C_m. We also represent every method $m \in \mathsf{dom}(\Gamma)$ by the CFG $(\{v_m\}, \emptyset, v_m, v_m)$, which corresponds to a method body that returns immediately after having been called. This CFG does not have any edges, because in the client-local semantics, we do not execute the implementation of such a method, but use its specification to incorporate the effect a call to the method has on the program state. Finally, if $\mathcal{P}$ does not have a client (so $n = 0$), we define a CFG of the form $(\{v^t_{\mathsf{mgc}}\}, \emptyset, v^t_{\mathsf{mgc}}, v^t_{\mathsf{mgc}})$ for each thread $t \in \mathsf{ThreadID}$, and let $N_0 = \{v^t_{\mathsf{mgc}} \mid t \in \mathsf{ThreadID}\}$. These CFGs are used to represent the most general client of the methods appearing in Γ'; see below. If $\mathcal{P}$ contains a client, we let $N_0 = \emptyset$.

We often view the above collection of CFGs as a single graph with the node set $\mathsf{Node} = N_0 \uplus \biguplus_{t=1}^{n} N_t \uplus \biguplus_{m \in \mathsf{Method}(\mathcal{P}) \uplus \mathsf{dom}(\Gamma)} N_m$ and the edge set $T = \biguplus_{t=1}^{n} T_t \uplus \biguplus_{m \in \mathsf{Method}(\mathcal{P}) \uplus \mathsf{dom}(\Gamma)} T_m$.

Operational semantics with ownership transfer. Consider a program $\Gamma \vdash \mathcal{P} : \Gamma'$

represented by its CFG. Let Pos be the set of ***thread positions***:

$$\mathsf{Pos} = \mathsf{Node} \uplus (\mathsf{Node} \times \mathsf{Node}).$$

A thread position describes the call-stack of a thread: its last component describes the program point of the current command, and the other component, when it exists, the return point for the method called.

We define the set of program configurations as

$$\mathsf{Config} = (\mathsf{ThreadID} \rightharpoonup_{fin} \mathsf{Pos}) \times \Sigma \cup \{\top\}.$$

The special configuration $\top$ indicates an error. The first component of a non-erroneous configuration is a program counter, which defines the position of each thread in the program, and the second defines the state of the program memory.

The operational semantics of $\Gamma \vdash \mathcal{P} : \Gamma'$ is given by the transition relation $\longrightarrow_{\Gamma,\mathcal{P},\Gamma'}$: $\mathsf{Config} \times \mathsf{Act} \times \mathsf{Config}$ in Figure 4. The rules package a semantics for complete and open programs into a single transition relation.

Equivalence of the semantics. Our operational semantics induces the trace interpretation of programs $\Gamma \vdash \mathcal{P} : \Gamma'$. For a finite trace τ and $\sigma, \sigma' \in \mathsf{Config}$ we write $\sigma \xrightarrow{\tau}{}^*_{\Gamma,\mathcal{P},\Gamma'} \sigma'$ if there exists a corresponding derivation of τ using $\longrightarrow_{\Gamma,\mathcal{P},\Gamma'}$. We denote by PC_0 the set of initial program counters of $\mathcal{P}$, which is $\{[1 : \mathsf{start}_1, \ldots, n : \mathsf{start}_n]\}$ when $\mathcal{P}$ contains a client and $\{[1 : v^1_{\mathsf{mgc}}, \ldots, n : v^n_{\mathsf{mgc}}] \mid n \geq 1\}$ when it does not. The trace interpretation of $\mathcal{P}$ is defined as follows: $[\![\Gamma \vdash \mathcal{P} : \Gamma']\!]_{\mathsf{op}}\theta = \top$, if $\exists \tau, \mathsf{pc}_0 \in \mathsf{PC}_0.\, (\mathsf{pc}_0, \theta) \xrightarrow{\tau}{}^*_{\Gamma,\mathcal{P},\Gamma'} \top$; otherwise,

$$[\![\Gamma \vdash \mathcal{P} : \Gamma']\!]_{\mathsf{op}}\theta = \{\tau \mid \exists \mathsf{pc}_0 \in \mathsf{PC}_0.\, \exists \sigma \in \mathsf{Config} - \{\top\}.\, (\mathsf{pc}_0, \theta) \xrightarrow{\tau}{}^*_{\Gamma,\mathcal{P},\Gamma'} \sigma\}.$$

The following is a variation on the standard result about the equivalence of the operational and the denotation semantics of a while-language.

Lemma 23. $[\![\Gamma \vdash \mathcal{P} : \Gamma']\!]_{\mathsf{op}}\theta = [\![\Gamma \vdash \mathcal{P} : \Gamma']\!]\theta$.

Auxiliary definitions. We define several operations that relate configurations in the semantics of $\mathcal{C}(\mathcal{L})$, $\mathcal{C}$ and $\mathcal{L}$. A partial operation $* : \mathsf{Pos} \times \mathsf{Pos} \rightharpoonup \mathsf{Pos}$ combines thread positions in the semantics of $\mathcal{C}$ and $\mathcal{L}$ to obtain a position of $\mathcal{C}(\mathcal{L})$ as follows: $v * v^t_{\mathsf{mgc}} = v$ and $(v, v_m) * (v^t_{\mathsf{mgc}}, v') = (v, v')$ for $t \in \mathsf{ThreadID}$. For all the other combinations, the $*$ operator is undefined. We lift $*$ to program counters pointwise and define $*$ on non-erroneous configurations from Config:

$$(\mathsf{pc}_1, \theta_1) * (\mathsf{pc}_2, \theta_2) = (\mathsf{pc}_1 * \mathsf{pc}_2, \theta_1 * \theta_2).$$

For a set of methods M we define functions $\mathsf{client} : \mathsf{Pos} \to \mathsf{Pos}$ and $\mathsf{lib}^t : \mathsf{Pos} \to \mathsf{Pos}$, $t \in \mathsf{ThreadID}$ that compute thread positions in the semantics of $\mathcal{C}$ and $\mathcal{L}$ corresponding to a position in $\mathcal{C}(\mathcal{L})$. We let $\mathsf{client}(v) = v$, $\mathsf{lib}(v) = v^t_{\mathsf{mgc}}$, $\mathsf{client}(v, v') = (v, v_m)$, $\mathsf{lib}(v, v') = (v^t_{\mathsf{mgc}}, v')$. We then lift these operations to program counters as follows: $(\mathsf{client}(\mathsf{pc}))(t) = \mathsf{client}(\mathsf{pc}(t))$ and $(\mathsf{lib}(\mathsf{pc}))(t) = \mathsf{lib}(\mathsf{pc}(t))$.

In the following, we denote the set of all call actions by CallAct, return actions with RetAct, and the actions of both kinds by CallRetAct.

$$\frac{(\rho, c, \rho') \in T \cup \{((v_0, v), c, (v_0, v')) \mid (v, c, v') \in T\} \qquad f_c^t(\theta) \neq \top \qquad \theta' \in f_c^t(\theta)}{\mathsf{pc}[t : \rho], \theta \xrightarrow{(t,c)} \mathsf{pc}[t : \rho'], \theta'} \tag{1}$$

$$\frac{(\rho, c, \rho') \in T \cup \{((v_0, v), c, (v_0, v')) \mid (v, c, v') \in T\} \quad f_c^t(\theta) = \top}{\mathsf{pc}[t : \rho], \theta \xrightarrow{(t,c)} \top} \tag{2}$$

Complete programs:

$$\frac{(v, m, v') \in T}{\mathsf{pc}[t : v], \theta \xrightarrow{(t,\mathsf{call}\ m)} \mathsf{pc}[t : (v', \mathsf{start}_m)], \theta} \tag{3}$$

$$\frac{}{\mathsf{pc}[t : (v, \mathsf{end}_m)], \theta \xrightarrow{(t,\mathsf{ret}\ m)} \mathsf{pc}[t : v], \theta} \tag{4}$$

Client programs:

$$\frac{(v, m, v') \in T \qquad \{p\}\, m\, \{q\} \in \Gamma \qquad (\theta \setminus p_t)\!\downarrow}{\mathsf{pc}[t : v], \theta \xrightarrow{(t,\mathsf{call}\ m(\theta_p))} \mathsf{pc}[t : (v, v_m)], \theta \setminus p_t} \tag{5}$$

$$\frac{(v, m, v') \in T \qquad \{p\}\, m\, \{q\} \in \Gamma \qquad (\theta \setminus p_t)\!\uparrow}{\mathsf{pc}[t : v], \theta \xrightarrow{(t,\mathsf{call}\ m(\epsilon))} \top} \tag{6}$$

$$\frac{\{p\}\, m\, \{q\} \in \Gamma \qquad \theta_q \in q_t \qquad (\theta * \theta_q)\!\downarrow}{\mathsf{pc}[t : (v, v_m)], \theta \xrightarrow{(t,\mathsf{ret}\ m(\theta_q))} \mathsf{pc}[t : v], \theta * \theta_q} \tag{7}$$

Library programs:

$$\frac{\{p\}\, m\, \{q\} \in \Gamma' \qquad \theta_p \in p_t \qquad (\theta * \theta_p)\!\downarrow}{\mathsf{pc}[t : v^t_{\mathsf{mgc}}], \theta \xrightarrow{(t,\mathsf{call}\ m(\theta_p))} \mathsf{pc}[t : (v^t_{\mathsf{mgc}}, \mathsf{start}_m)], \theta * \theta_p} \tag{8}$$

$$\frac{\{p\}\, m\, \{q\} \in \Gamma' \qquad (\theta \setminus q_t)\!\downarrow}{\mathsf{pc}[t : (v^t_{\mathsf{mgc}}, \mathsf{end}_m)], \theta \xrightarrow{(t,\mathsf{ret}\ m(\theta_q))} \mathsf{pc}[t : v^t_{\mathsf{mgc}}], \theta \setminus q_t} \tag{9}$$

$$\frac{\{p\}\, m\, \{q\} \in \Gamma' \qquad (\theta \setminus q_t)\!\uparrow}{\mathsf{pc}[t : (v^t_{\mathsf{mgc}}, \mathsf{end}_m)], \theta \xrightarrow{(t,\mathsf{ret}\ m(\epsilon))} \top} \tag{10}$$

Fig. 4. Transition relation $\longrightarrow_{\Gamma,\mathcal{P},\Gamma'}$ for a specified program $\Gamma \vdash \mathcal{P} : \Gamma'$. We omit the subscripts $\Gamma, \mathcal{P}, \Gamma'$ to avoid clutter.

B.2 Properties of the $\circ$ and $\setminus$ Operations on Footprints

Proposition 24. *The $\circ$ operation is well-defined.*

Proof. Consider $l_1, l_2 \in \mathcal{F}(\Sigma)$ and $\theta_1, \theta_2 \in \Sigma$ such that $l_1 = \delta(\theta_1)$ and $l_2 = \delta(\theta_2)$. Take another pair of states $\theta_1', \theta_2' \in \Sigma$ such that $l_1 = \delta(\theta_1')$ and $l_2 = \delta(\theta_2')$.

Thus, $\theta_1' \in \delta(\theta_1)$ and $\theta_2' \in \delta(\theta_2)$, which implies:

$$(\theta_1' * \theta_2')\downarrow \iff (\theta_1 * \theta_2')\downarrow \iff (\theta_1 * \theta_2)\downarrow.$$

Furthermore, if $\theta_1' * \theta_2'$ and $\theta_1 * \theta_2$ are defined, then for all $\theta' \in \Sigma$,

$$(\theta_1' * \theta_2' * \theta')\downarrow \iff (\theta_1 * \theta_2' * \theta')\downarrow \iff (\theta_1 * \theta_2 * \theta')\downarrow.$$

Hence, $\delta(\theta_1 * \theta_2) = \delta(\theta_1' * \theta_2')$, so that $\circ$ is well-defined. □

Proposition 25. *The $\setminus$ operation is well-defined.*

Proof. Consider $l_1, l_2 \in \mathcal{F}(\Sigma)$ and $\theta_1, \theta_2, \theta_1', \theta_2', \theta, \theta' \in \Sigma$ such that $\theta_1, \theta_1' \in l_1$, $\theta_2, \theta_2' \in l_2$, $\theta_1 = \theta_2 * \theta$ and $\theta_1' = \theta_2' * \theta'$. We have:

$$\delta(\theta_2 * \theta) = \delta(\theta_1) = l_1 = \delta(\theta_1') = \delta(\theta_2' * \theta').$$

Since $\delta(\theta_2) = \delta(\theta_2')$, by Definition 3 this implies $\delta(\theta) = \delta(\theta')$, so that $\setminus$ is well-defined. □

We now list some the useful properties of $\setminus$. In the following, we use the equality that means both sides are defined and equal, or both are undefined.

Proposition 26. *For all $l_1, l_2, l_3 \in \mathcal{F}(\Sigma)$, we have:*

1. $(l_1 \setminus l_2) \circ l_2 = l_1$, *if $l_1 \setminus l_2$ is defined;*
2. $(l_1 \circ l_2) \setminus l_2 = l_1$, *if $l_1 \circ l_2$ is defined;*
3. $(l_1 \circ l_2) \setminus l_3 = (l_1 \setminus l_3) \circ l_2$, *if $l_1 \circ l_2$ and $l_1 \setminus l_3$ are defined.*

Proof. 1. Consider $\theta \in l_1 \setminus l_2$. Then there exist $\theta_1 \in l_1$ and $\theta_2 \in l_2$ such that $\theta * \theta_2 = \theta_1$. Hence,

$$(l_1 \setminus l_2) \circ l_2 = \delta(\theta * \theta_2) = \delta(\theta_1) = l_1.$$

2. Since $l_1 \circ l_2$ is defined, so is $\theta_1 * \theta_2$ for some $\theta_1 \in l_1$ and $\theta_2 \in l_2$. But then $\theta_1 \in (l_1 \circ l_2) \setminus l_2$.

3. By item 1, we get

$$l_1 \circ l_2 = ((l_1 \setminus l_3) \circ l_3) \circ l_2 = ((l_1 \setminus l_3) \circ l_2) \circ l_3.$$

Hence, $(l_1 \circ l_2) \setminus l_3 = (l_1 \setminus l_3) \circ l_2$. □

B.3 Proof of Lemma 13

Below we sometimes write $\sqsubseteq_\pi$ instead of $\sqsubseteq$ to make the bijection π used to establish the relation between histories explicit.

Take $\theta, \theta' \in \Sigma$ and consider a trace ξ' executable from θ'. Assume histories H, H' such that $\mathsf{history}(\xi') = H'$, $H \sqsubseteq H'$ and H is balanced from $l_1 = \delta(\theta)$. By Proposition 11, H' is balanced from $l_2 = \delta(\theta')$. We prove that there exists a trace ξ executable from θ' such that $\mathsf{history}(\xi) = H$.

To this end, we define a finite sequence of steps that transforms ξ' into ξ. In more detail, ξ is constructed using a sequence of traces α_k executable from θ', defined for every prefix H_k of H of length k as described below. Every trace α_k is such that for some prefix β_k of α_k we have $\mathsf{history}(\beta_k) = H_k$ and $H \sqsubseteq_\pi \mathsf{history}(\alpha_k)$, where π is an identity on actions in H_k. Additionally, for all i, j with $i < j$, β_i is a prefix of β_j. Hence, the sequence of traces β_k has a limit ξ such that for every k, β_k is a prefix of ξ and $\mathsf{history}(\xi) = H$. Then, as we show, ξ is executable from θ' and $\mathsf{history}(\xi) = H$.

The transformation constructing α_{k+1} from α_k has one delicate case, whose proof relies on the rest of transformation. We now formulate an auxiliary notion used in this case, motivated by the proof of Lemma 28 below.

Definition 27. *Two histories*

$$H^1 = H(t_1, \mathsf{call}\ m_1(\theta_1))H_1(t_2, \mathsf{ret}\ m_2(\theta_2))H_2,$$
$$H^2 = HH'_1(t_2, \mathsf{ret}\ m_2(\theta_2))(t_1, \mathsf{call}\ m_1(\theta_1))H'_2$$

*are **conflicting** if*

- *H^1 and H^2 are balanced from some l_1 and l_2, respectively, such that $l_2 \preceq l_1$;*
- *$t_1 \neq t_2$;*
- *$H^1 \sqsubseteq_\pi H^2$;*
- *π is an identity on actions in H and maps the $(t_1, \mathsf{ret}\ m_1(\theta_1))$ and $(t_2, \mathsf{call}\ m_2(\theta_2))$ actions in H^1 to the corresponding actions shown in H^2;*
- *the history*

$$HH'_1(t_1, \mathsf{call}\ m_1(\theta_1))(t_2, \mathsf{ret}\ m_2(\theta_2))H'_2 \tag{11}$$

  is not balanced from l_2.

To construct the sequence of traces, we let $\alpha_0 = \xi$ and let the prefix β_0 contain all the client actions preceding the first call or return action in α_0. The trace α_{k+1} is constructed from the trace α_k by applying the following lemma for $\xi_1 = \beta_k$, $\xi_1\xi_2 = \alpha_k$, $H'_1 = H_k$ and $H'_1\psi H'_2 = H$. The lemma either successfully constructs α_{k+1}, or yields a pair of conflicting histories. In the following, we use techniques developed in the proof of the lemma to show that no two conflicting histories can exist, which means that the transformation succeeds in all cases, thus establishing the required.

Lemma 28. *Consider a history $H'_1\psi H'_2$, and a library trace $\xi_1\xi_2$ executable from θ' such that*

$$\mathsf{history}(\xi_1) = H'_1, \tag{12}$$
$$H'_1\psi H'_2 \sqsubseteq_\pi \mathsf{history}(\xi_1\xi_2), \tag{13}$$

where π is an identity on actions in H'_1. Then either $H'_1\psi H'_2$ and another history composed of actions from $\mathsf{history}(\xi_1\xi_2)$ are conflicting, or there exist traces ξ'_2 and ξ''_2 such that $\xi_1\xi'_2\xi''_2$ is executable from θ' and

$$\mathsf{history}(\xi_1\xi'_2) = H'_1\psi, \tag{14}$$
$$H'_1\psi H'_2 \sqsubseteq_{\pi'} \mathsf{history}(\xi_1\xi'_2\xi''_2), \tag{15}$$

where π' is an identity on actions in $H'_1\psi$ in $\mathsf{history}(\xi_1\xi'_2\xi''_2)$.

To prove Lemma 28, we transform $\xi_1\xi_2$ into $\xi_1\xi'_2\xi''_2$ by applying a finite number of transformations that preserve their properties of interest, described in Propositions 29–31 below.

Proposition 29. *Let ξ be a library trace and H a history such that $H \sqsubseteq_\pi \mathsf{history}(\xi)$. Then swapping any two adjacent actions $\varphi_1\varphi_2$ in ξ executed by different threads such that*

1. $\varphi_2 \in \mathsf{CallAct}$ and if $\varphi_1 \in \mathsf{RetAct}$, then φ_2 precedes φ_1 in H; or
2. $\varphi_1 \in \mathsf{RetAct}$ and $\varphi_2 \in \mathsf{Act} - \mathsf{CallAct}$,

yields a trace ξ' such that $H \sqsubseteq_{\pi'} \mathsf{history}(\xi')$.

The bijection π' is defined as follows. If $\varphi_1 \notin \mathsf{CallRetAct}$ or $\varphi_2 \notin \mathsf{CallRetAct}$, then $\pi' = \pi$. Otherwise, let i be the index of φ_1 in $\mathsf{history}(\xi)$. Then $\pi'(i+1) = \pi(i)$, $\pi'(i) = \pi(i+1)$ and $\pi'(k) = \pi(k)$ for $k \notin \{i, i+1\}$.

Proposition 30. *Swapping any two adjacent actions $\varphi_1\varphi_2$ in a library trace ξ such that*

– *$\varphi_1 \in \mathsf{Act} - \mathsf{RetAct}$, $\varphi_2 \in \mathsf{CallAct}$; or*
– *$\varphi_1 \in \mathsf{RetAct}$, $\varphi_2 \in \mathsf{Act} - \mathsf{CallAct}$,*

preserves the executability of ξ from a given state.

Proof. Consider $\xi = \xi_1\varphi_1\varphi_2\xi_2$ and a state θ_0 such that ξ is executable from θ_0. The proof proceeds by case analysis on the kind of actions φ_1 and φ_2. We consider only one illustrative case, where $\varphi_1 = (t_1, c)$ and $\varphi_2 = (t_2, \mathsf{call}\ m_2(\theta))$. In this case, $f_c^{t_2}([\![\xi_1]\!]_{\mathsf{lib}}\theta_0) \neq \top$ and

$$[\![\xi_1\varphi_1\varphi_2]\!]_{\mathsf{lib}}\theta_0 = f_c^{t_2}([\![\xi_1]\!]_{\mathsf{lib}}\theta_0) * \{\theta\} \neq \emptyset.$$

By the Footprint Preservation property, we get $(([\![\xi_1]\!]_{\mathsf{lib}}\theta_0) * \theta)\downarrow$. Then by the Strong Locality property we get

$$[\![\xi_1\varphi_1\varphi_2]\!]_{\mathsf{lib}}\theta_0 = f_c^{t_2}([\![\xi_1]\!]_{\mathsf{lib}}\theta_0) * \{\theta\} = f_c^{t_2}([\![\xi_1]\!]_{\mathsf{lib}}\theta_0 * \{\theta\}) = [\![\xi_1\varphi_2\varphi_1]\!]_{\mathsf{lib}}\theta_0.$$

□

Note that the above proposition does not allow swapping a return followed by a call. This case is very subtle and relies crucially on the balancedness property of histories under consideration. The swapping can be done under a certain condition, which is in fact fulfilled in the context where we apply it in the proof of Lemma 28.

Proposition 31. *Consider a library trace*

$$\xi = \xi_1(t_2, \mathsf{ret}\ m_2(\theta_2))(t_1, \mathsf{call}\ m_1(\theta_1))\xi_2$$

executable from θ and a trace

$$\xi' = \xi_1(t_1, \mathsf{call}\ m_1(\theta_1))(t_2, \mathsf{ret}\ m_2(\theta_2))\xi_2.$$

If $\mathsf{history}(\xi')$ is balanced from $\delta(\theta)$, then ξ' is also executable from θ.

Proof of Lemma 28. From (12) and (13) it follows that $\xi_2 = \xi_3'\psi\xi_4'$ for some traces ξ_3' and ξ_4'. We have two cases.

1. $\psi \in \mathsf{CallAct}$. Let t be the thread executing ψ. From (12) and (13), any return action by t preceding ψ in $\mathsf{history}(\xi_1\xi_2)$ is in H_1', and thus not in ξ_3'. Furthermore, the thread that executed ψ does not execute any actions in the subtrace ξ_3'. Thus, we can try to move the action ψ to the beginning of ξ_3' by swapping adjacent actions in $\xi_1\xi_2$ a

finite number of times as described in Propositions 29(1), 30 and 31. If we succeed, we obtain the trace $\xi_1\psi\xi_3'\xi_4'$. Conditions (14)–(15) then follow from Proposition 29(1) for $\xi_2' = \psi$ and $\xi_2'' = \xi_3'\xi_4'$. We can only fail when Proposition 31 is not applicable due to the history of the target trace ξ' not being balanced. In this case, $H_1'\psi H_2'$ and the history of the source trace ξ in Proposition 31 are conflicting.

2. $\psi \in \mathsf{RetAct}$. Then ξ_3' cannot contain an action $\varphi \in \mathsf{CallAct}$, because in this case φ would precede ψ in $\mathsf{history}(\xi_1\xi_2)$. However, φ is in H_2' and, thus, ψ precedess φ in $H_1'\psi H_2'$, so this would contradict (13). Hence, there are no call actions in ξ_3'. Moreover, for any action $\varphi = (t, \mathsf{ret}\ \_)$ in ξ_3' there are no actions by the thread t in ξ_3' following φ. Thus, we can move all actions from RetAct in the subtrace ξ_3' of $\xi_1\xi_2$ to the position right after ψ by swapping adjacent actions in $\xi_1\xi_2$ a finite number of times as described in Propositions 29(2) and 30. We thus obtain the trace $\xi_1\xi_3''\psi\xi_4''\xi_4'$, where ξ_4'' consists of return actions in ξ_3' and ξ_3'' of the rest of actions in the subtrace. Conditions (14)–(15) then follow from Proposition 29(2) for $\xi_2' = \xi_3''\psi$ and $\xi_2'' = \xi_4''\xi_4'$. □

All that remains is to show that there are no conflicting histories, hence guaranteeing that the transformation in Proposition 31 is always applicable in the proof of Lemma 28. To this end, we use the following auxiliary proposition.

Proposition 32. *Assume H is identical to H', except it has extra calls, and H and H' are balanced from l_1 and l_2, respectively, such that $l_2 \preceq l_1$. Then the $\circ$-combination l_c of footprints of states transferred at the extra call actions in H is defined, H' is balanced from l_1 and*

$$[\![H]\!]^\sharp l_1 = ([\![H']\!]^\sharp l_1) \circ l_c \ \wedge\ [\![H']\!]^\sharp l_2 \preceq [\![H']\!]^\sharp l_1.$$

Proof. We prove the required by induction on the length of H. If H is empty, then so is H' and $l_c = \delta(\varepsilon)$. Assume the statement of the proposition is valid for all histories H of length less than $n > 0$. Consider a history $H = H_0\varphi$ of length n and a corresponding history H' satisfying the conditions in the proposition. We now make a case split on the type of the action φ.

– φ is a call transferring θ_0 that is not in H'. Then H_0 and H' are identical except H_0 has extra calls. Hence, by the induction hypothesis for H_0 and H', H' is balanced from l_1, $[\![H']\!]^\sharp l_2 \preceq [\![H']\!]^\sharp l_1$ and

$$[\![H]\!]^\sharp l_1 = [\![H_0\varphi]\!]^\sharp l_1 = ([\![H_0]\!]^\sharp l_1) \circ \delta(\theta_0) = ([\![H'^\sharp]\!] l_1) \circ (l_c \circ \delta(\theta_0)).$$

– φ is a call transferring θ_0 also present in H'. Then $H' = H_0'\varphi$, where H_0' and H_0 are identical except H_0 has extra calls. Hence, by the induction hypothesis for H_0 and H_0', we have

$$[\![H]\!]^\sharp l_1 = [\![H_0\varphi]\!]^\sharp l_1 = ([\![H_0]\!]^\sharp l_1) \circ \delta(\theta_0) = \\ ([\![H_0']\!]^\sharp l_1) \circ l_c \circ \delta(\theta_0) = ([\![H_0'\varphi]\!]^\sharp l_1) \circ \delta(\theta_0) = ([\![H']\!]^\sharp l_1) \circ \delta(\theta_0).$$

In particular, H' is balanced from l_1. By the induction hypothesis for H_0 and H_0', we also have $[\![H_0']\!]^\sharp l_2 \preceq [\![H_0']\!]^\sharp l_1$. From this we get

$$[\![H']\!]^\sharp l_2 = [\![H_0'\varphi]\!]^\sharp l_2 = ([\![H_0']\!]^\sharp l_2) \circ \delta(\theta_0) \preceq \\ ([\![H_0']\!]^\sharp l_1) \circ \delta(\theta_0) = [\![H_0'\varphi]\!]^\sharp l_1 = [\![H']\!]^\sharp l_1.$$

– φ is a return transferring θ_0. Then it is also present in H', so that $H' = H'_0\varphi$, where H_0 and H'_0 are identical except H_0 has extra calls. Then by the induction hypothesis for H_0 and H'_0, we have:

$$[\![H]\!]^\sharp l = [\![H_0\varphi]\!]^\sharp l_1 = ([\![H_0]\!]^\sharp l_1) \setminus \delta(\theta_0) = (([\![H'_0]\!]^\sharp l_1) \circ l_c) \setminus \delta(\theta_0).$$

Since $H' = H'_0\varphi$ is balanced from l_2, $([\![H'_0]\!]^\sharp l_2) \setminus \theta_0$ is defined. Furthermore, by the induction hypothesis for H_0 and H'_0, we also have $[\![H'_0]\!]^\sharp l_2 \preceq [\![H'_0]\!]^\sharp l_1$. Hence, $([\![H'_0]\!]^\sharp l_1) \setminus \theta_0$ is defined as well. By Proposition 26(3), we then have:

$$[\![H]\!]^\sharp l_1 = (([\![H'_0]\!]^\sharp l_1) \circ l_c) \setminus \delta(\theta_0) = \\ (([\![H'_0]\!]^\sharp l_1) \setminus \delta(\theta_0)) \circ l_c = ([\![H'_0\varphi]\!]^\sharp l_1) \circ l_c = ([\![H']\!]^\sharp l_1) \circ l_c.$$

In particular, H' is balanced from l_1. From $[\![H'_0]\!]^\sharp l_2 \preceq [\![H'_0]\!]^\sharp l_1$, it also follows that

$$[\![H']\!]^\sharp l_2 = [\![H'_0\varphi]\!]^\sharp l_2 = ([\![H'_0]\!]^\sharp l_2) \setminus \delta(\theta_0) \preceq \\ ([\![H'_0]\!]^\sharp l_1) \setminus \delta(\theta_0) = [\![H'_0\varphi]\!]^\sharp l_1 = [\![H']\!]^\sharp l_1.$$

□

Lemma 33. *There are no conflicting pairs of histories.*

Proof. Consider histories H^1 and H^2 satisfying the conditions in Definition 27. We recall the form of the histories:

$$H^1 = H(t_1, \mathsf{call}\ m_1(\theta_1))H_1(t_2, \mathsf{ret}\ m_2(\theta_2))H_2,$$
$$H^2 = HH'_1(t_2, \mathsf{ret}\ m_2(\theta_2))(t_1, \mathsf{call}\ m_1(\theta_1))H'_2.$$

Since H^2 is balanced from l_2,

$$[\![HH'_1(t_2, \mathsf{ret}\ m_2(\theta_2))]\!]^\sharp l_2 = ([\![HH'_1]\!]^\sharp l_2) \setminus \delta(\theta_2)$$

is defined. Assume

$$[\![HH'_1(t_1, \mathsf{call}\ m_1(\theta_1))]\!]^\sharp l_2 = ([\![HH'_1]\!]^\sharp l_2) \circ \delta(\theta_1)$$

is defined. Since $([\![HH'_1]\!]^\sharp l_2) \setminus \delta(\theta_2)$ is defined, by Proposition 26(3), we have:

$$\begin{aligned} &[\![HH'_1(t_1, \mathsf{call}\ m_1(\theta_1))(t_2, \mathsf{ret}\ m_2(\theta_2))]\!]^\sharp l_2 \\ &= (([\![HH'_1]\!]^\sharp l_2) \circ \delta(\theta_1)) \setminus \delta(\theta_2) \\ &= (([\![HH'_1]\!]^\sharp l_2) \setminus \delta(\theta_2)) \circ \delta(\theta_1) \\ &= [\![HH'_1(t_2, \mathsf{ret}\ m_2(\theta_2))(t_1, \mathsf{call}\ m_1(\theta_1))]\!]^\sharp l_2. \end{aligned}$$

But then (11) is balanced from l_2, contradicting our assumptions. Hence, $(([\![HH'_1]\!]^\sharp l_2) \circ \delta(\theta_1))\uparrow$.

A call action in H'_1 cannot be in H_2: in this case it would follow $(t_2, \mathsf{ret}\ m_2(\theta_2))$ in H^1, but precede it in H^2, contradicting $H^1 \sqsubseteq H^2$. Hence, all call actions in H'_1 are in H_1. Let $H_1 = H_3H_4$, where H_3 is the minimal prefix of H_1 containing all call actions from H'_1. Then

$$H^1 = H(t_1, \mathsf{call}\ m_1(\theta_1))H_3H_4(t_2, \mathsf{ret}\ m_2(\theta_2))H_2.$$

If H_3 is non-empty, any return action in it precedes its last call action, which is also in H_1'. Since $H^1 \sqsubseteq_\pi H^2$, such a return action also has to be in H_1'. Thus, all return actions in H_3 are in H_1'. Furthermore, any action by thread t_1 in H^1 preceding $(t_1, \mathsf{call}\ m_1(\theta_1))$ has to be in H. Since linearizability preserves the order of actions by the same thread, this means that t_1 does not execute any actions in H_1'.

The traces H^1 and H^2 are of the following more general form, which we denote (**F**):

$$H^1 = H_0 H_3 H_4 (t_2, \mathsf{ret}\ m_2(\theta_2)) H_2,$$
$$H^2 = H_0' H_1' (t_2, \mathsf{ret}\ m_2(\theta_2))(t_1, \mathsf{call}\ m_1(\theta_1)) H_2',$$

where

- H^1 and H^2 are balanced from some l_1 and l_2, respectively, such that $l_2 \preceq l_1$;
- H_0 and H_0' are identical, except H_0 has some extra call actions;
- all call actions in H_1' are in H_3;
- all return actions in H_3 are in H_1';
- $H^1 \sqsubseteq_\pi H^2$;
- $(t_1, \mathsf{call}\ m_1(\theta_1))$ is in H_0;
- π maps actions in H_0' to those in H_0, $(t_1, \mathsf{call}\ m_1(\theta_1))$ to the corresponding acton in H_0, and $(t_2, \mathsf{ret}\ m_2(\theta_2))$ to the same action shown in H^1;
- for all actions $(t, \mathsf{call}\ \_)$ in H_3 that are not in H_1', t does not execute any actions in H_1'; and
- $((\llbracket H_0' H_1' \rrbracket^\sharp l_2) \circ \delta(\theta_1))\uparrow$.

This is obtained by letting $H_0 = H(t_1, \mathsf{call}\ m_1(\theta_1))$ and $H_0' = H'$.

In the following, we describe a process that transforms the histories H^1 and H^2 into another pair of histories satisfying the conditions above, but such that H_3 is strictly smaller. Repeatedly applying this process, we can make H_3 empty, obtaining histories satisfying (**F**):

$$H_0 H_4 (t_2, \mathsf{ret}\ m_2(\theta_2)) H_2,$$
$$H_0' H_1' (t_2, \mathsf{ret}\ m_2(\theta_2))(t_1, \mathsf{call}\ m_1(\theta_1)) H_2'.$$

In particular, $((\llbracket H_0' H_1' \rrbracket^\sharp l_2) \circ \delta(\theta_1))\uparrow$.

Since all calls from H_1' are in $H_3 = \varepsilon$, H_1' contains only returns. Since the histories are balanced from l_1 and l_2, respectively, $\llbracket H_0 \rrbracket^\sharp l_1$ and $\llbracket H_0' \rrbracket^\sharp l_2$ are defined. The history H_0 is identical to H_0', except it has extra calls. By Proposition 32, the $\circ$-combination of footprints of states transferred at the extra call actions in H_0 is defined. Since the $(t_1, \mathsf{call}\ m_1(\theta_1))$ action is in H_0, but not in H_0', this combination is thus of the form $\delta(\theta_1) \circ l_c$ for some l_c and

$$\llbracket H_0 \rrbracket^\sharp l_1 = (\llbracket H_0' \rrbracket^\sharp l_1) \circ \delta(\theta_1) \circ l_c \ \wedge\ \llbracket H_0' \rrbracket^\sharp l_2 \preceq \llbracket H_0' \rrbracket^\sharp l_1.$$

Hence, $(\llbracket H_0' \rrbracket^\sharp l_2) \circ \delta(\theta_1)$ is defined. Since H_1' contains only return actions,

$$\llbracket H_0' H_1' \rrbracket^\sharp l_2 = (\llbracket H_0' \rrbracket^\sharp l_2) \setminus l',$$

where l' is the $\circ$-combination of the footprints of states transferred at these actions. This implies

$$\llbracket H_0' \rrbracket^\sharp l_2 = (\llbracket H_0' H_1' \rrbracket^\sharp l_2) \circ l',$$

where both expressions are defined. But then so is

$$([\![H_0']\!]l_2)^\sharp \circ \delta(\theta_1) = ([\![H_0'H_1']\!]^\sharp l_2) \circ l' \circ \delta(\theta_1).$$

Hence, $([\![H_0'H_1']\!]^\sharp l_2) \circ \delta(\theta_1)$ is defined, contradicting the opposite fact established above. This contradiction implies that a conflicting pair of histories does not exist.

Assume histories H^1 and H^2 satisfying (**F**):

$$H^1 = H_0H_3H_4(t_2, \mathsf{ret}\ m_2(\theta_2))H_2,$$
$$H^2 = H_0'H_1'(t_2, \mathsf{ret}\ m_2(\theta_2))(t_2, \mathsf{call}\ m_1(\theta_1))H_2',$$

We now show that from these we can construct another pair of histories satisfying (**F**), but with H_3 strictly smaller. We make a case split on the next action in H_3.

- $H_3 = (t, \mathsf{call}\ m(\theta))H_5$, such that the call action is not in H_1'. Since linearizability preserves the order of actions by the same thread, t does not execute any actions in H_1'. Then we can let $H_0 := H_0(t, \mathsf{call}\ m(\theta))$ and $H_3 := H_5$.
- $H_3 = (t, \mathsf{call}\ m(\theta))H_5$, such that the call action is in H_1'. Let $H_1' = H_3'(t, \mathsf{call}\ m(\theta))H_4'$, then

  $$H^1 = H_0(t, \mathsf{call}\ m(\theta))H_5H_4(t_2, \mathsf{ret}\ m_2(\theta_2))H_2,$$
  $$H^2 = H_0'H_3'(t, \mathsf{call}\ m(\theta))H_4'(t_2, \mathsf{ret}\ m_2(\theta_2))(t_2, \mathsf{call}\ m_1(\theta_1))H_2'.$$

  Using the transformations from item 1 in the proof of Lemma 28, we can try to move the action $(t, \mathsf{call}\ m(\theta))$ to the beginning of H_3'. If this succeeds, we can let $H_0 := H_0(t, \mathsf{call}\ m(\theta))$, $H_3 := H_5$, $H_0' := H_0'(t, \mathsf{call}\ m(\theta))$ and $H_1' := H_3'H_4'$. Otherwise, we get a pair of conflicting histories, which are of the form (**F**) but with a smaller H_3.
- $H_3 = (t, \mathsf{ret}\ m(\theta))H_5$. Then the return action is also in H_1', so that $H_1' = H_3'(t, \mathsf{ret}\ m(\theta))H_4'$:

  $$H^1 = H_0(t, \mathsf{ret}\ m(\theta))H_5H_4(t_2, \mathsf{ret}\ m_2(\theta_2))H_2,$$
  $$H^2 = H_0'H_3'(t, \mathsf{ret}\ m(\theta))H_4'(t_2, \mathsf{ret}\ m_2(\theta_2))(t_2, \mathsf{call}\ m_1(\theta_1))H_2'.$$

  Using the transformations from item 2 in the proof of Lemma 28, we can move the return action to the beginning of H_3', obtaining a pair of histories:

  $$H^1 = H_0(t, \mathsf{ret}\ m(\theta))H_5H_4(t_2, \mathsf{ret}\ m_2(\theta_2))H_2,$$
  $$H^2 = H_0'(t, \mathsf{ret}\ m(\theta))H_3'H_4'(t_2, \mathsf{ret}\ m_2(\theta_2))(t_2, \mathsf{call}\ m_1(\theta_1))H_2'.$$

  Then we can let $H_0 := H_0(t, \mathsf{ret}\ m(\theta))$, $H_3 := H_5$, $H_0' := H_0'(t, \mathsf{ret}\ m(\theta))$ and $H_1' := H_3'H_4'$.

□

B.4 Proof of Lemma 16

Consider $\theta_1^0 \in I_1$ and $\theta_2^0 \in I_2$ and let

$$\sigma^0 = (\mathsf{pc}_0, \theta_1^0 * \theta_2^0) \ \wedge\ \sigma_1^0 = (\mathsf{client}(\mathsf{pc}_0), \theta_1^0) \ \wedge\ \sigma_2^0 = (\mathsf{lib}(\mathsf{pc}_0), \theta_2^0),$$

where $\mathsf{PC}_0 = \{\mathsf{pc}_0\}$. Then $\sigma^0 = \sigma_1^0 * \sigma_2^0$.

Take a τ-labelled derivation using $\longrightarrow_{\mathcal{C}(\mathcal{L})}$ that starts from $\sigma^0 \in \mathsf{Config}$. From this, we now construct traces η and ξ, together with their derivations using $\longrightarrow_{\mathcal{C}}$ and $\longrightarrow_{\mathcal{L}}$, such that if τ leads to $\top$, then so does η or ξ, and, otherwise,

$$\eta \in [\![\mathcal{C}]\!]\theta_1^0 \wedge \xi \in [\![\mathcal{L}]\!]\theta_2^0 \wedge \mathsf{history}(\eta) = \mathsf{history}(\xi) \wedge \mathsf{client}(\tau) = \mathsf{ground}(\eta) \wedge \mathsf{lib}(\tau) = \mathsf{ground}(\xi).$$

Our construction first considers every prefix τ_i of τ and builds traces η_i and ξ_i in the semantics of $\mathcal{C}$ and $\mathcal{L}$ and their derivations for τ_i. The two resulting series are such that for $i < j$, the derivation of η_i or ξ_i is a prefix of that of η_j or ξ_j, which also implies that the trace η_i or ξ_i is a prefix of η_j or ξ_j. Because of this, the series have the limit derivations and the limit traces, which are the desired ones.

The following claim lies at the core of our construction:

Consider a prefix τ_i of τ, traces η_i and ξ_i and configurations $\sigma, \sigma_1, \sigma_2 \in \mathsf{Config}$ such that $\sigma = \sigma_1 * \sigma_2 \neq \top$,

$$\sigma^0 \xrightarrow{\tau_i}{}^*_{\mathcal{C}(\mathcal{L})} \sigma \ \wedge\ \sigma_1^0 \xrightarrow{\eta_i}{}^*_{\mathcal{C}} \sigma_1 \ \wedge\ \sigma_2^0 \xrightarrow{\xi_i}{}^*_{\mathcal{L}} \sigma_2$$

and

$$\mathsf{history}(\eta) = \mathsf{history}(\xi) \wedge \mathsf{client}(\tau_i) = \mathsf{ground}(\eta_i) \wedge \mathsf{lib}(\tau_i) = \mathsf{ground}(\xi_i).$$

Assume $\tau = \tau_i \varphi \tau'$ for some action φ and trace τ'. If $\sigma \xrightarrow{\varphi}_{\mathcal{C}(\mathcal{L})} \sigma'$ for some $\sigma' \neq \top$, then there exist $\sigma_1', \sigma_2' \in \mathsf{Config} - \{\top\}$ and extensions η_{i+1} and ξ_{i+1} of η_i and ξ_i with the corresponding derivations such that $\sigma' = \sigma_1' * \sigma_2'$,

$$\sigma_1^0 \xrightarrow{\eta_{i+1}}{}^*_{\mathcal{C}} \sigma_1' \quad \wedge \quad \sigma_2^0 \xrightarrow{\xi_{i+1}}{}^*_{\mathcal{L}} \sigma_2'$$

and

$$\mathsf{history}(\eta_{i+1}) = \mathsf{history}(\xi_{i+1}) \wedge \mathsf{client}(\tau_{i+1}) = \mathsf{ground}(\eta_{i+1}) \wedge \mathsf{lib}(\tau_{i+1}) = \mathsf{ground}(\xi_{i+1}).$$

Also, if $\sigma \xrightarrow{\varphi}_{\mathcal{C}(\mathcal{L})} \top$, then there exist extensions η_{i+1} and ξ_{i+1} of η_i and ξ_i such that

$$\sigma_1^0 \xrightarrow{\eta_{i+1}}{}^*_{\mathcal{C}} \top \quad \vee \quad \sigma_2^0 \xrightarrow{\xi_{i+1}}{}^*_{\mathcal{L}} \top.$$

To prove the claim, we assume $\tau_i, \varphi, \tau', \sigma, \sigma_1, \sigma_2, \sigma', \eta_i, \xi_i$ satisfying the assumptions in it. We now make a case-split on which of the rules (1)–(10) is used to derive φ.

– Rule (1) such that $\varphi = (t, c) \in \mathsf{PAct}$ is a client action in τ. Then, there exist pc, v, v', $\theta, \theta', \theta_1, \theta_2$, such that (v, c, v') is in the control-flow relation of $\mathcal{C}$, $\mathsf{pc}(t)\uparrow$ and

$$\sigma_1 = (\mathsf{client}(\mathsf{pc})[t : v], \theta_1) \ \wedge\ \sigma_2 = (\mathsf{lib}(\mathsf{pc})[t : v^t_{\mathsf{mgc}}], \theta_2) \ \wedge\ \sigma = (\mathsf{pc}[t : v], \theta) \ \wedge\ \theta = \theta_1 * \theta_2 \tag{16}$$

and

$$\sigma' = (\mathsf{pc}[t : v'], \theta') \ \wedge\ \theta' \in f_c^t(\theta).$$

Then by the Strong Locality property, we have

$$\theta' \in f_c^t(\theta) = f_c^t(\theta_1 * \theta_2) = f_c^t(\theta_1) * \{\theta_2\}.$$

Thus, $\theta' = \theta'_1 * \theta_2$, for some $\theta'_1 \in f_c^t(\theta_1)$. Let

$$\sigma'_1 = (\mathsf{client}(\mathsf{pc})[t : v'], \theta'_1),$$

$\sigma'_2 = \sigma_2$, $\eta_{i+1} = \eta_i\varphi$ and $\xi_{i+1} = \xi_i$. Then $\sigma' = \sigma'_1 * \sigma'_2$ and

$$\sigma_1 \xrightarrow{\varphi}_{\mathcal{C}} \sigma'_1.$$

We also have

$$\mathsf{client}(\tau_i\varphi) = \mathsf{client}(\tau_i)\varphi = \mathsf{ground}(\eta_i)\varphi = \mathsf{ground}(\eta_i\varphi) = \mathsf{ground}(\eta_{i+1}),$$

$$\mathsf{lib}(\tau_i\varphi) = \mathsf{lib}(\tau_i) = \mathsf{ground}(\xi_i) = \mathsf{ground}(\xi_{i+1})$$

and

$$\mathsf{history}(\eta_{i+1}) = \mathsf{history}(\eta_i) = \mathsf{history}(\xi_i) = \mathsf{history}(\eta_{i+1}),$$

from which the required follows.

– Rule (2) such that $\varphi = (t, c) \in \mathsf{PAct}$ is a client action in τ. Then there exist pc, v, v', $\theta, \theta_1, \theta_2$, such that (v, m, v') is in the control-flow relation of $\mathcal{C}$, (16) is fulfilled, $\mathsf{pc}(t)\uparrow$ and $f_c^t(\theta) = \top$. By the Strong Locality property, this implies $f_c^t(\theta_1) = \top$. But then

$$\sigma_1 \xrightarrow{\varphi}_{\mathcal{C}} \top,$$

as required.

– Rules (1) or (2) such that $\varphi \in \mathsf{PAct}$ is a library action in τ. This case is handled similarly to the previous two.

– Rule (3) such that $\varphi = (t, \mathsf{call}\ m)$. Then there exist pc, v, v', $\theta, \theta_1, \theta_2$, such that (v, m, v') is in the control-flow relation of $\mathcal{C}$, (16) is fulfilled, $\mathsf{pc}(t)\uparrow$ and

$$\sigma' = (\mathsf{pc}[t : (v', \mathsf{start}_m)], \theta).$$

Since $\mathcal{C}$ is safe, we have $\theta_1 = \theta'_1 * \theta_p$ for some $\theta_p \in p_t$, where $\Gamma(m) = (p, \_)$. Let

$$\sigma'_1 = (\mathsf{client}(\mathsf{pc})[t : (v', v_m)]) \ \wedge\ \sigma'_2 = (\mathsf{lib}(\mathsf{pc})[t : (v^t_{\mathsf{mgc}}, \mathsf{start}_m)]),$$

so that $\sigma'_1 * \sigma'_2 = \sigma'$ and

$$\sigma_1 \xrightarrow{(t,\mathsf{call}\ m(\theta_p))}_{\mathcal{C}} \sigma'_1 \ \wedge\ \sigma_2 \xrightarrow{(t,\mathsf{call}\ m(\theta_p))}_{\mathcal{L}} \sigma'_2.$$

Now let $\eta_{i+1} = \eta_i(t, \mathsf{call}\ m(\theta_p))$ and $\xi_{i+1} = \xi_i(t, \mathsf{call}\ m(\theta_p))$, then $\mathsf{history}(\eta_{i+1}) = \mathsf{history}(\xi_{i+1})$,

$$\mathsf{client}(\tau_i\varphi) = \mathsf{client}(\tau_i)\varphi = \mathsf{ground}(\eta_i)\varphi = \mathsf{ground}(\eta_i(t, \mathsf{call}\ m(\theta_p))) = \mathsf{ground}(\eta_{i+1}),$$

$$\mathsf{lib}(\tau_i\varphi) = \mathsf{lib}(\tau_i)\varphi = \mathsf{ground}(\xi_i)\varphi = \mathsf{ground}(\xi_i(t, \mathsf{call}\ m(\theta_p))) = \mathsf{ground}(\xi_{i+1})$$

and

$$\mathsf{history}(\eta_{i+1}) = \mathsf{history}(\eta_i\varphi) = \mathsf{history}(\eta_i)\varphi = \mathsf{history}(\xi_i)\varphi = \mathsf{history}(\xi_i\varphi) = \mathsf{history}(\eta_{i+1}).$$

Hence, the required follows.

– Rule (4) such that $\varphi = (t, \mathsf{ret}\ m)$. Then there exist pc, v', $\theta, \theta_1, \theta_2$, such that $\mathsf{pc}(t)\uparrow$ and

$$\begin{aligned}\sigma = (\mathsf{pc}[t : (v', \mathsf{end}_m)], \theta)\ \wedge\ \sigma' = (\mathsf{pc}[t : v'], \theta)\ \wedge\ \theta = \theta_1 * \theta_2\\ \wedge\ \sigma_1 = (\mathsf{client}(\mathsf{pc})[t : (v', v_m)], \theta_1)\\ \wedge\ \sigma_2 = (\mathsf{lib}(\mathsf{pc})[t : v^t_{\mathsf{mgc}}\mathsf{end}_m], \theta_2).\end{aligned}$$

Since $\mathcal{L}$ is safe, we have $\theta_2 = \theta_2' * \theta_q$ for some $\theta_q \in q_t$, where $\Gamma(m) = (\_, q)$. Hence, for

$$\sigma_1' = (\mathsf{client}(\mathsf{pc})[t : v'], \theta_1 * \theta_q)\ \wedge\ \sigma_2' = (\mathsf{lib}(\mathsf{pc})[t : v^t_{\mathsf{mgc}}], \theta_2'),$$

we have $\sigma_1' * \sigma_2' = \sigma'$ and

$$\sigma_1 \xrightarrow{(t, \mathsf{ret}\ m(\theta_q))}_{\mathcal{C}} \sigma_1'\ \wedge\ \sigma_2 \xrightarrow{(t, \mathsf{ret}\ m(\theta_q))}_{\mathcal{L}} \sigma_2'.$$

Let $\eta_{i+1} = \eta_i(t, \mathsf{ret}\ m(\theta_q))$ and $\xi_{i+1} = \xi_i(t, \mathsf{ret}\ m(\theta_q))$, then the proof is finished like in the previous case.

We have thus shown that the claim holds for all φ. □

B.5 Proof of Lemma 17

Assume $\eta \in [\![\mathcal{C}]\!]\theta_1$ and $\xi \in [\![\mathcal{L}]\!]\theta_2$ such that $\mathsf{history}(\eta) = \mathsf{history}(\xi)$ and $(\theta_1 * \theta_2)\downarrow$. Then there exist an η-labelled derivation using $\longrightarrow_{\mathcal{C}}$ and a ξ-labelled derivation using $\longrightarrow_{\mathcal{L}}$, starting from initial configurations $\sigma_0^1 \in \mathsf{Config}$ and $\sigma_0^2 \in \mathsf{Config}$, respectively. Without loss of generality, we can assume that σ_0^1 and σ_0^2 have the same number of threads.

From the above two derivations, we now construct the required trace $\tau \in [\![\mathcal{C}(\mathcal{L})]\!](\theta_1 * \theta_2)$ together with its derivation using $\longrightarrow_{\mathcal{C}(\mathcal{L})}$. We first build a series of traces

$$\tau_0, \tau_1, \tau_2, \ldots$$

and their derivations. This series is such that for $i < j$, the derivation of τ_i is a prefix of that of τ_j, which also implies that τ_i is a prefix of τ_j. Because of this, the series has the limit derivation and the limit trace, which are the desired ones.

The first element in the series is the empty trace ε and the empty derivation consisting of the initial configuration $\sigma_0^1 * \sigma_0^2$ only. For the $(i{+}1)$-st element with $i > 0$, we assume that the i-th element τ_i and its computation have been constructed and satisfy the following property:

For some prefixes η_1 and ξ_1 of η and ξ such that

$$\mathsf{history}(\eta_1) = \mathsf{history}(\xi_1)\ \wedge\ \mathsf{client}(\tau_i) = \mathsf{ground}(\eta_1)\ \wedge\ \mathsf{lib}(\tau_i) = \mathsf{ground}(\xi_1)$$

and configurations $\sigma_i^1, \sigma_i^2 \in \mathsf{Config}$ we have:

$$\sigma_0^1 * \sigma_0^2 \xrightarrow{\tau_i}{}^*_{\mathcal{C}(\mathcal{L})} \sigma_i^1 * \sigma_i^2\ \wedge\ \sigma_0^1 \xrightarrow{\eta_1}{}^*_{\mathcal{C}} \sigma_i^1\ \wedge\ \sigma_0^2 \xrightarrow{\xi_1}{}^*_{\mathcal{L}} \sigma_i^2.$$

Now we define the $(i+1)$-st element τ_{i+1} and its derivation that maintain the property above. As we explained above, the derivation for τ_{i+1} will be an extension of that for τ_i by one or more steps.

Assume $\eta = \eta_1\varphi_1\eta'$ and $\xi = \xi_1\varphi_2\xi'$ for some actions φ_1 and φ_2 and traces η' and ξ' (the case of $\eta = \eta_1$ or $\xi = \xi_1$ is handled analogously). Let the following be the transitions by φ_1 and φ_2 in the derivations for η and ξ:

$$\sigma_i^1 \xrightarrow{\varphi_1}_{\mathcal{C}} \sigma^1 \quad \wedge \quad \sigma_i^2 \xrightarrow{\varphi_2}_{\mathcal{L}} \sigma^2,$$

where $\sigma^1, \sigma^2 \in \mathsf{Config} - \{\top\}$. We now make a case-split on types of actions φ_1 and φ_2 and which of the rules (1)–(10) are used to derive them.

– $\varphi_1, \varphi_2 \in \mathsf{CallRetAct}$ derived using rules (5) and (8) or (7) and (9). In this case $\varphi_1 = \varphi_2$, because $\mathsf{history}(\eta_1)$ and $\mathsf{history}(\xi_1)$ are the same by our assumption, so that their same-length prefixes $\mathsf{history}(\tau_i)\varphi_1$ and $\mathsf{history}(\tau_i)\varphi_2$ have to be identical. We only consider the case of $\varphi_1, \varphi_2 \in \mathsf{CallAct}$; the case of $\varphi_1, \varphi_2 \in \mathsf{RetAct}$ is handled similarly. Assume $\varphi_1 = (t, \mathsf{call}\ m(\theta_p))$ for $\Gamma(m) = (p, \_)$ and $\theta_p \in p_t$. Then, there exist pc_1, pc_2, v, v', θ_1, θ_2, such that (v, m, v') is in the control-flow relation of $\mathcal{C}$, $\mathsf{pc}_1(t)\uparrow$, $\mathsf{pc}_2(t)\uparrow$ and

$$\begin{aligned}
&\sigma_i^1 = (\mathsf{pc}_1[t : v], \theta_1)\\
\wedge\ &\sigma^1 = (\mathsf{pc}_1[t : (v', v_m)], \theta_1')\\
\wedge\ &\sigma_i^2 = (\mathsf{pc}_2[t : v^t_{\mathsf{mgc}}], \theta_2)\\
\wedge\ &\sigma^2 = (\mathsf{pc}_2[t : (v^t_{\mathsf{mgc}}, \mathsf{start}_m)], \theta_2 * \theta_p)\\
\wedge\ &\sigma_i^1 * \sigma_i^2 = ((\mathsf{pc}_1 * \mathsf{pc}_2)[t : v], \theta_1 * \theta_2)\ \wedge\ \theta_1 = \theta_1' * \theta_p.
\end{aligned}$$

In this case

$$\sigma^1 * \sigma^2 = ((\mathsf{pc}_1 * \mathsf{pc}_2)[t : (v', \mathsf{start}_m)], \theta_1 * \theta_2),$$

is defined. Then $\sigma^1 * \sigma^2$ is defined as well and

$$\sigma_i^1 * \sigma_i^2 \xrightarrow{(t, \mathsf{call}\ m)}_{\mathcal{C}(\mathcal{L})} \sigma^1 * \sigma^2.$$

Hence, the desired τ_{i+1} is $\tau_i\varphi_1$, and its derivation is

$$\sigma_0^1 * \sigma_0^2 \xrightarrow{\tau_i}{}^*_{\mathcal{C}(\mathcal{L})} \sigma_i^1 * \sigma_i^2 \xrightarrow{(t, \mathsf{call}\ m)}_{\mathcal{C}(\mathcal{L})} \sigma^1 * \sigma^2.$$

– $\varphi_2 = (t, c) \in \mathsf{PAct}$, derived using (1). In this case for some pc_1, pc_2, $\theta_1, \theta_2, \theta_2'$, v, v', we have that (v, c, v') is in the control-flow relation of $\mathcal{L}$, $\mathsf{pc}_1(t)\uparrow$, $\mathsf{pc}_2(t)\uparrow$ and

$$\begin{aligned}
&\sigma_i^1 = (\mathsf{pc}_1[t : (v_0, v_m)], \theta_1)\ \wedge\\
&\sigma_i^2 = (\mathsf{pc}_2[t : (v^t_{\mathsf{mgc}}, v)], \theta_2)\ \wedge\\
&\sigma^2 = (\mathsf{pc}_2[t : (v^t_{\mathsf{mgc}}, v')], \theta_2')\ \wedge\ \theta_2' \in f_c^t(\theta_2)\ \wedge\\
&\sigma_i^1 * \sigma_i^2 = ((\mathsf{pc}_1 * \mathsf{pc}_2)[t : (v_0, v)], \theta_1 * \theta_2).
\end{aligned}$$

Since $\theta_1 * \theta_2$ is defined, by the Footprint Preservation property, $\theta_2' \in f_c^t(\theta_2)$ implies that $\theta_1 * \theta_2'$ is defined as well. Then

$$\sigma_i^1 * \sigma^2 = ((\mathsf{pc}_1 * \mathsf{pc}_2)[t : (v_0, v')], \theta_1 * \theta_2').$$

is defined. From the Strong Locality property, we have

$$\theta_1 * \theta_2' \in \{\theta_1\} * f_c^t(\theta_2) = f_c^t(\theta_1 * \theta_2).$$

Hence, $\sigma_i^1 * \sigma_i^2 \xrightarrow{\varphi_2}_{\mathcal{C}(\mathcal{L})} \sigma_i^1 * \sigma^2$. The desired τ_{i+1} is $\tau_i\varphi_2$, and its derivation is:

$$\sigma_0^1 * \sigma_0^2 \xrightarrow{\tau_i}{}^*_{\mathcal{C}(\mathcal{L})} \sigma_i^1 * \sigma_i^2 \xrightarrow{\varphi_2}_{\mathcal{C}(\mathcal{L})} \sigma_i^1 * \sigma^2.$$

– $\varphi_1 \in \mathsf{PAct}$. This case is handled similarly to previous one.

We have just shown how to construct τ_i for all the cases. The desired derivation is constructed as the limit of the sequence for τ_i. It is easy to show that the resulting trace τ satisfies $\mathsf{client}(\tau) = \mathsf{ground}(\eta)$ and $\mathsf{lib}(\tau) = \mathsf{ground}(\xi)$. □

B.6 Proof of Corollary 20

For a configuration $\sigma = (\mathsf{pc}, \_)$, let $\sigma(t) = \mathsf{pc}(t)$.

Consider states $\theta, \theta' \in I$ and histories H, H' such that $(\delta(\theta), H) \sqsubseteq (\delta(\theta'), H')$ and $\mathcal{L}$ is safe at θ'. Take a trace $\xi' \in [\![\mathcal{L}]\!]\theta'$ such that $\mathsf{history}(\xi') = H'$. By Lemma 13, there exists a trace ξ executable from θ' such that $\mathsf{history}(\xi) = H$ and $\xi \sim \xi'$. We now show that $\xi \in [\![\mathcal{L}]\!]\theta'$.

Let $\xi' = \varphi_1\varphi_2\varphi_3 \ldots$ and $\xi = \lambda_1\lambda_2\lambda_3 \ldots$. Let the following be a derivation of ξ' in the operational semantics of $\mathcal{L}$:

$$\sigma_0 \xrightarrow{\varphi_1}_{\mathcal{L}} \sigma_1 \xrightarrow{\varphi_2}_{\mathcal{L}} \sigma_2 \xrightarrow{\varphi_3}_{\mathcal{L}} \sigma_3 \xrightarrow{\varphi_4}_{\mathcal{L}} \ldots$$

Using this derivation, we construct a series of finite derivations and define the desired derivation for ξ as the limit of this series.

The starting point of the series is the empty sequence σ_0. To construct the $(i+1)$-st element of the series, we make the following assumptions about the i-th one:

1. The i-th computation sequence has the form
$$\sigma'_0 \xrightarrow{\lambda_1}_{\mathcal{L}} \sigma'_1 \xrightarrow{\lambda_2}_{\mathcal{L}} \sigma'_2 \xrightarrow{\lambda_3}_{\mathcal{L}} \ldots \xrightarrow{\lambda_i}_{\mathcal{L}} \sigma'_i,$$
where $\sigma'_0 = \sigma_0$.
2. For all t and $j \le i$, if ξ'_1 is the minimal prefix of ξ' such that $\xi'_1|_t = (\lambda_1 \ldots \lambda_j)|_t$, then
$$\sigma_{|\xi'_1|}(t) = \sigma'_j(t).$$

Under these assumptions, we extend the given i-th computation sequence by the $(i+1)$-st action of η. The result of this extension becomes the $(i+1)$-st sequence, which as we show, meets the three assumptions described above.

Let λ_{i+1} be the $(i+1)$-st action of ξ. To find a desired computation sequence, we only need to find a client configuration σ'_{i+1} such that

$$\sigma'_i \xrightarrow{\lambda_{i+1}}_{\mathcal{L}} \sigma'_{i+1}, \tag{17}$$

and σ'_{i+1} satisfies condition 2 above.

Assume $\lambda_{i+1} = (t', \_)$. Let ξ'_1 be the minimal prefix of ξ' such that $\xi'_1|_{t'} = (\lambda_1 \ldots \lambda_i)|_{t'}$, ξ'_2, its maximal prefix such that $\xi'_2|_{t'} = (\lambda_1 \ldots \lambda_i)|_{t'}$, and ξ'_3, its minimal prefix such that $\xi'_3|_{t'} = (\lambda_1 \ldots \lambda_i\lambda_{i+1})|_{t'}$. By the induction hypothesis, $\sigma_{|\xi'_1|}(t') = \sigma'_i(t')$. Then $\sigma_{|\xi'_1|}(t') = \sigma_{|\xi'_2|}(t')$. Thus, $\sigma'_i(t') = \sigma_{|\xi'_2|}(t')$. Since λ_{i+1} is executable from $\sigma_{|\xi'_2|}$, and ξ is executable, from this we can easily show that (17) holds for some $\sigma'_{i+1} \neq \top$ such that $\sigma'_{i+1}(t') = \sigma_{|\xi'_3|}(t')$. We let the derivation for $\lambda_1 \ldots \lambda_i$ extended with this transition be the desired new derivation. Since $\sigma'_{i+1}(t') = \sigma_{|\xi'_3|}(t')$, this derivation satisfies condition 2 for $t = t'$. For $t \neq t'$, we have $\sigma'_{i+1}(t) = \sigma'_i(t)$. Hence, it also satisfies condition 2 for all $t \neq t'$. □

B.7 Proof of Theorem 22

Consider Γ and Γ' satisfying the conditions of the theorem. For a trace ξ obtained from a library satisfying Γ', we let $\lceil\!\lceil \xi \rceil\!\rceil_\Gamma$ be the trace ξ with every interface action φ replaced by the action $\lceil\!\lceil \varphi \rceil\!\rceil_\Gamma$ defined as follows:

$$\lceil\!\lceil (t, \mathsf{call}\ m(\theta * \theta')) \rceil\!\rceil_\Gamma = (t, \mathsf{call}\ m(\theta)),\ \text{if } \theta \in p_t^m;$$
$$\lceil\!\lceil (t, \mathsf{ret}\ m(\theta * \theta')) \rceil\!\rceil_\Gamma = (t, \mathsf{ret}\ m(\theta)),\ \text{if } \theta \in q_t^m.$$

Since pre- and postconditions in method specifications are precise, this operation is well-defined.

The proof of the theorem relies on the following lemmas, proved at the end of this section.

Lemma 34. *If $\xi \in [\![\mathcal{L} : \Gamma']\!](\theta_0 * \theta_0')$, $\mathcal{L} : \Gamma$ is safe at θ_0, and the trace $\lfloor\!\lfloor \mathsf{history}(\xi) \rfloor\!\rfloor_\Gamma$ is executable from θ_0', then $\lceil\!\lceil \xi \rceil\!\rceil_\Gamma \in [\![\mathcal{L} : \Gamma]\!]\theta_0$.*

Lemma 35. *Assume $\xi \in [\![\mathcal{L} : \Gamma]\!]\theta_0$, $(\theta_0 * \theta_0')\downarrow$, $\mathcal{L} : \Gamma'$ is safe at $\theta_0 * \theta_0'$, ξ is such that $\xi = \lceil\!\lceil \xi' \rceil\!\rceil_\Gamma$ and $\lfloor\!\lfloor \xi' \rfloor\!\rfloor_\Gamma$ is executable from θ_0', and ξ'' is such that $\mathsf{history}(\xi'') = \mathsf{history}(\xi')$ and ξ'' is executable from $\theta_0'' * \theta_0'$, where $\delta(\theta_0) \preceq \delta(\theta_0'')$. Then $\xi' \in [\![\mathcal{L} : \Gamma']\!](\theta_0 * \theta_0')$.*

Proof of Theorem 22. Consider a trace $\xi_1 \in [\![\mathcal{L}_1 : \Gamma']\!](\theta_1 * \theta)$, where $\theta_1 \in I_1$ and $\theta \in I$. By the assumption (iv) of the theorem, $\lfloor\!\lfloor \mathsf{history}(\xi_1) \rfloor\!\rfloor_\Gamma$ is executable from θ. Then by Lemma 34, $\lceil\!\lceil \xi_1 \rceil\!\rceil_\Gamma \in [\![\mathcal{L}_1 : \Gamma]\!]\theta_1$. Since $(\mathcal{L}_1 : \Gamma, I_1) \sqsubseteq (\mathcal{L}_2 : \Gamma, I_2)$, for some $\theta_2 \in I_2$ and $\xi_2 \in [\![\mathcal{L}_2 : \Gamma]\!]\theta_2$ we have $(\delta(\theta_1), \mathsf{history}(\lceil\!\lceil \xi_1 \rceil\!\rceil_\Gamma)) \sqsubseteq (\delta(\theta_2), \mathsf{history}(\xi_2))$. By Corollary 20, there exists $\xi_2' \in [\![\mathcal{L}_2 : \Gamma]\!]\theta_2$ such that $\mathsf{history}(\xi_2') = \mathsf{history}(\lceil\!\lceil \xi_1 \rceil\!\rceil_\Gamma)$. Let ξ_2'' be the trace ξ_2' with its interface actions replaced so that they form the history $\mathsf{history}(\xi_1)$. Then $\lceil\!\lceil \xi_2'' \rceil\!\rceil_\Gamma = \xi_2'$ and $\lfloor\!\lfloor \mathsf{history}(\xi_2'') \rfloor\!\rfloor_\Gamma = \lfloor\!\lfloor \mathsf{history}(\xi_1) \rfloor\!\rfloor_\Gamma$. Since $\delta(\theta_2) \preceq \delta(\theta_1)$, we have $(\theta_2 * \theta)\downarrow$. Hence, by Lemma 35, $\xi_2'' \in [\![\mathcal{L}_2 : \Gamma']\!](\theta_2 * \theta)$, from which the required follows. □

Proof of Lemma 34. Consider $\xi \in [\![\mathcal{L} : \Gamma']\!](\theta_0 * \theta_0')$ and assume that $\mathcal{L} : \Gamma$ is safe at θ_0. Then there exists a derivation of ξ in the semantics of $\mathcal{L} : \Gamma'$ starting from a configuration $\sigma_1^0 = (\mathsf{pc}_0, \theta_0 * \theta_0')$. Let $\sigma_2^0 = (\mathsf{pc}_0, \theta_0)$. Using the derivation of ξ, we construct a series of finite derivations and define the desired derivation for $\lceil\!\lceil \xi \rceil\!\rceil_\Gamma$ as the limit of this series.

The starting point of the series is the empty sequence σ_2^0. The following claim shows how to construct the $(i+1)$-st element of the series from the i-th one:

Assume that ξ_i is a finite prefix of ξ, and for some $\mathsf{pc}, \theta_1, \theta_2$, we have

$$\sigma_1^0 \xrightarrow{\xi_i}{}^*_{\mathcal{L}:\Gamma'} (\mathsf{pc}, \theta_1) \ \wedge\ \sigma_2^0 \xrightarrow{\lceil\!\lceil \xi_i \rceil\!\rceil_\Gamma}{}^*_{\mathcal{L}:\Gamma} (\mathsf{pc}, \theta_2)$$

and $\theta_1 = \theta_2 * \theta$ for some $\theta \in [\![\lfloor\!\lfloor \mathsf{history}(\xi_i) \rfloor\!\rfloor_\Gamma]\!]_{\mathsf{lib}}\theta_0'$. Let $\xi = \xi_i\varphi\zeta$ for some action φ and trace ζ. If for some pc', θ_1',

$$(\mathsf{pc}, \theta_1) \xrightarrow{\varphi}_{\mathcal{L}:\Gamma'} (\mathsf{pc}', \theta_1'),$$

then there exists θ_2' such that

$$(\mathsf{pc}, \theta_2) \xrightarrow{\lceil\!\lceil \varphi \rceil\!\rceil_\Gamma}_{\mathcal{L}:\Gamma} (\mathsf{pc}', \theta_2')$$

and $\theta_1' = \theta_2' * \theta'$ for some $\theta' \in [\![\lfloor\!\lfloor \mathsf{history}(\xi_i\varphi) \rfloor\!\rfloor_\Gamma]\!]_{\mathsf{lib}}\theta_0'$.

To show the claim, we consider three cases, depending on the type of the action φ.

– $\varphi = (t, c)$. Since $\mathcal{L} : \Gamma$ is safe at θ_0, $f_c^t(\theta_2) \neq \top$. Hence, by the Strong Locality property, we have

$$\theta_1' \in f_c^t(\theta_1) = f_c^t(\theta_2 * \theta) = f_c^t(\theta_2) * \{\theta\},$$

so that $\theta_1' = \theta_2' * \theta$ for some $\theta_2' \in f_c^t(\theta_2)$. Besides,

$$\theta \in [\![\lfloor\!\lfloor \mathsf{history}(\xi_i) \rfloor\!\rfloor_\Gamma]\!]_{\mathsf{lib}} \theta_0' = [\![\lfloor\!\lfloor \mathsf{history}(\xi_i \varphi) \rfloor\!\rfloor_\Gamma]\!]_{\mathsf{lib}} \theta_0'.$$

Thus, the required holds for $\theta' = \theta$.

– $\varphi = (t, \mathsf{call}\ m(\theta_p * \theta_p'))$, where $\theta_p \in p_t^m$ and $\theta_p * \theta_p' \in r_t^m$. Then $\theta_1' = \theta_1 * \theta_p * \theta_p' = (\theta_2 * \theta_p) * (\theta * \theta_p')$. Besides,

$$\theta * \theta_p' \in ([\![\lfloor\!\lfloor \mathsf{history}(\xi_i) \rfloor\!\rfloor_\Gamma]\!]_{\mathsf{lib}} \theta_0') * \{\theta_p'\} = [\![\lfloor\!\lfloor \mathsf{history}(\xi_i \varphi) \rfloor\!\rfloor_\Gamma]\!]_{\mathsf{lib}} \theta_0'.$$

Hence, the required holds for $\theta_2' = \theta_2 * \theta_p$ and $\theta' = \theta * \theta_p'$.

– $\varphi = (t, \mathsf{ret}\ m(\theta_q * \theta_q'))$, where $\theta_q \in q_t^m$ and $\theta_q * \theta_q' \in s_t^m$. Then $\theta_2 * \theta = \theta_1 = \theta_1' * \theta_q * \theta_q'$. Since $\mathcal{L} : \Gamma$ is safe at θ_0, $\theta_2 = \theta_2' * \theta_q$ for some θ_2', so that $\theta_2' * \theta_q * \theta = \theta_1' * \theta_q * \theta_q'$. Since $*$ is cancellative, this entails $\theta_2' * \theta = \theta_1' * \theta_q'$. The trace $\lfloor\!\lfloor \mathsf{history}(\xi_i \varphi) \rfloor\!\rfloor_\Gamma$ is executable from θ_0' and $\theta \in [\![\lfloor\!\lfloor \mathsf{history}(\xi_i) \rfloor\!\rfloor_\Gamma]\!]_{\mathsf{lib}} \theta_0'$. Hence, $\theta \setminus \theta_q'$ is defined and $\theta_1' = \theta_2' * (\theta \setminus \theta_q')$. Thus, the required holds for $\theta' = \theta \setminus \theta_q'$.

□

Proof of Lemma 35. Consider $\xi \in [\![\mathcal{L} : \Gamma]\!]\theta_0$ and ξ' such that $\xi = [\![\xi']\!]_\Gamma$ and $\lfloor\!\lfloor \mathsf{history}(\xi') \rfloor\!\rfloor_\Gamma$ is executable from θ_0'. Further, assume ξ'' such that $\mathsf{history}(\xi'') = \mathsf{history}(\xi')$ and ξ'' is executable from $\theta_0'' * \theta_0'$ for $\delta(\theta_0) \preceq \delta(\theta_0'')$. There exists a derivation of ξ in the semantics of $\mathcal{L} : \Gamma$ starting from a configuration $\sigma_1^0 = (\mathsf{pc}_0, \theta_0)$. Let $\sigma_2^0 = (\mathsf{pc}_0, \theta_0 * \theta_0')$. Using the derivation of ξ, we construct a series of finite derivations and define the desired derivation for ξ' as the limit of this series.

The starting point of the series is the empty sequence σ_2^0. The following claim shows how to construct the $(i+1)$-st element of the series from the i-th one:

Assume that ξ_i is a finite prefix of ξ, and ξ_i' is the corresponding prefix of ξ'. Furthermore, for some $\mathsf{pc}, \theta_1, \theta_2$, we have

$$\sigma_1^0 \xrightarrow{\xi_i}{}^*_{\mathcal{L}:\Gamma} (\mathsf{pc}, \theta_1) \ \wedge \ \sigma_2^0 \xrightarrow{\xi_i'}{}^*_{\mathcal{L}:\Gamma'} (\mathsf{pc}, \theta_2)$$

and $\theta_2 = \theta_1 * \theta$ for some $\theta \in [\![\lfloor\!\lfloor \mathsf{history}(\xi_i') \rfloor\!\rfloor_\Gamma]\!]_{\mathsf{lib}} \theta_0'$. Let $\xi = \xi_i \varphi \zeta$ and $\xi' = \xi_i' \varphi' \zeta'$ for some actions φ, φ' and traces ζ, ζ'. If for some pc', θ_1',

$$(\mathsf{pc}, \theta_1) \xrightarrow{\varphi}_{\mathcal{L}:\Gamma'} (\mathsf{pc}', \theta_1'),$$

then there exists θ_2' such that

$$(\mathsf{pc}, \theta_2) \xrightarrow{\varphi'}_{\mathcal{L}:\Gamma} (\mathsf{pc}', \theta_2')$$

and $\theta_2' = \theta_1' * \theta'$ for some $\theta' \in [\![\lfloor\!\lfloor \mathsf{history}(\xi_i' \varphi') \rfloor\!\rfloor_\Gamma]\!]_{\mathsf{lib}} \theta_0'$.

To show the claim, we consider three cases, depending on the type of the action φ.

– $\varphi = (t, c)$. Then $\theta'_1 \in f^t_c(\theta_1)$. Since $(\theta_1 * \theta)\downarrow$, by the Footprint Preservation property, $(\theta'_1 * \theta)\downarrow$. Then by the Strong Locality property,

$$\theta'_1 * \theta \in f^t_c(\theta_1) * \{\theta\} = f^t_c(\theta_1 * \theta) = f^t_c(\theta_2).$$

Besides,

$$\theta \in [\![\lfloor\!\lfloor \mathsf{history}(\xi_i) \rfloor\!\rfloor_\Gamma]\!]_{\mathsf{lib}} \theta'_0 = [\![\lfloor\!\lfloor \mathsf{history}(\xi_i \varphi) \rfloor\!\rfloor_\Gamma]\!]_{\mathsf{lib}} \theta'_0.$$

Hence, the required holds for $\theta' = \theta$ and $\theta'_2 = \theta'_1 * \theta$.

– $\varphi = (t, \mathsf{call}\ m(\theta_p))$, where $\theta_p \in p^m_t$. Then $\varphi' = (t, \mathsf{call}\ m(\theta_p * \theta'_p))$ for some θ'_p such that $\theta_p * \theta'_p \in r^m_t$. In this case we have $\theta'_1 = \theta_1 * \theta_p$. Let ξ''_i be the minimal prefix of ξ'' such that $\mathsf{history}(\xi''_i) = \mathsf{history}(\xi'_i)$. Then there exists $\theta_3 \in [\![\xi''_i]\!]_{\mathsf{lib}}(\theta''_0 * \theta'_0)$ we have $(\theta_3 * \theta_p * \theta'_p)\downarrow$. Since $\delta(\theta_0) \preceq \delta(\theta''_0)$ and $\theta_2 \in [\![\xi'_i]\!]_{\mathsf{lib}}(\theta_0 * \theta'_0)$, from the Footprint Preservation and Strong Locality properties, it easily follows that $\delta(\theta_2) \preceq \delta(\theta_3)$. Hence, $(\theta_2 * \theta_p * \theta'_p)\downarrow$.
Then

$$\theta_2 * \theta_p * \theta'_p = \theta_1 * \theta * \theta_p * \theta'_p = \theta'_1 * (\theta * \theta'_p).$$

Besides,

$$\theta * \theta'_p \in ([\![\lfloor\!\lfloor \mathsf{history}(\xi'_i) \rfloor\!\rfloor_\Gamma]\!]_{\mathsf{lib}} \theta'_0) * \{\theta'_p\} = [\![\lfloor\!\lfloor \mathsf{history}(\xi'_i \varphi') \rfloor\!\rfloor_\Gamma]\!]_{\mathsf{lib}} \theta'_0.$$

Hence, the required holds for $\theta' = \theta * \theta'_p$ and $\theta'_2 = \theta_2 * \theta_p * \theta'_p$.

– $\varphi = (t, \mathsf{ret}\ m(\theta_q))$, where $\theta_q \in q^m_t$. Then $\varphi' = (t, \mathsf{ret}\ m(\theta_q * \theta'_q))$ for some θ'_q such that $\theta_q * \theta'_q \in s^m_t$. In this case we have $\theta'_1 = \theta_1 \setminus \theta_q$. Since $\lfloor\!\lfloor \mathsf{history}(\xi') \rfloor\!\rfloor_\Gamma$ is executable from θ'_0 and $\theta \in [\![\lfloor\!\lfloor \mathsf{history}(\xi'_i) \rfloor\!\rfloor_\Gamma]\!]_{\mathsf{lib}} \theta'_0$, we have $(\theta \setminus \theta'_q)\downarrow$. Since, $\theta_1 * \theta$ is defined, so is

$$\theta'_1 * (\theta \setminus \theta'_q) = (\theta_1 \setminus \theta_q) * (\theta \setminus \theta'_q) = (\theta_1 * \theta) \setminus (\theta_q * \theta'_q) = \theta_2 \setminus (\theta_q * \theta'_q).$$

Hence, the required holds for $\theta' = \theta \setminus \theta'_q$ and $\theta'_2 = \theta'_1 * (\theta \setminus \theta'_q)$.

□

C Logic for Safety

In this section we review an existing program logic that can be used to reason about open programs of Section 6. In Appendix D we extend the logic presented here for establishing the notion of linearizability proposed in Section 3.

Proving the safety of open programs is convenient in separation logics [14], because of their ability to reason naturally about ownership transfer. To deal with algorithms of the kind present in modern concurrent libraries, we need a logic that can handle programs with a high degree of interference between concurrent threads. For this reason, we use RGSep [16], which combines rely-guarantee (aka assume-guarantee) reasoning[2] with separation logic [14].

The main idea of the logic is to partition the program memory into several thread-local parts (each of which can only be accessed by a given thread) and the shared part (which can be accessed by all threads). The partitioning is defined by proofs in the

[2] A. Pnueli. In transition from global to modular temporal reasoning about programs. In *Logics and Models of Concurrent Systems*, 1985.

C. B. Jones. Specification and Design of (Parallel) Programs. In *IFIP Congress*, 1983.

logic: an assertion in the code of a thread restricts its local state and the shared state. Additionally, the partitioning is dynamic, meaning that we can use ownership transfer to move some part of the local state into the shared state and vice versa. Rely and guarantee conditions are then specified with sets of actions, which are relations *on the shared state* determining how threads can change it. This is in contrast with the original rely-guarantee method, in which rely and guarantee conditions are relations *on the whole program state*. Thus, while reasoning about a thread, we do not have to consider local states of other threads.

We present the logic in an abstract form [5], i.e., without fixing the underlying separation algebra Σ of memory states. Also, the variant of the logic we present here includes logical variables from a set $\mathsf{LVar} = \mathsf{LIVar} \uplus \mathsf{LSVar}$, where variables from $\mathsf{LIVar} = \{x, y, \ldots\}$ range over integers, and those from $\mathsf{LSVar} = \{X, Y, \ldots\}$, over memory states. Let $\mathsf{LVal} = \Sigma \cup \mathbb{Z}$ be the set of values of logical variables, and LInt, the set of their interpretations from $\mathsf{LVar} \to \mathsf{LVal}$ respecting the types.

We assume an assertion language for denoting subsets of $\Sigma \times \mathsf{LInt}$, including at least the following connectives:

$$p, q \ ::= \ \mathsf{true} \mid X \mid \exists X.\, p \mid \neg p \mid p \wedge q \mid p \vee q \mid p \Rightarrow q \mid \mathsf{emp} \mid p * q \mid p \mathrel{-\!\!*} q$$

The interpretation of most of them is standard. Therefore, we only give the most interesting cases:

$$\begin{array}{ll}
\theta, \mathbf{i} \models X & \iff \theta = \mathbf{i}(X) \\
\theta, \mathbf{i} \models \exists X.\, p & \iff \exists \theta' \in \Sigma.\, (\theta, \mathbf{i}[X : \theta'] \models p) \\
\theta, \mathbf{i} \models \exists x.\, p & \iff \exists u \in \mathbb{Z}.\, (\theta, \mathbf{i}[x : u] \models p) \\
\theta, \mathbf{i} \models \mathsf{emp} & \iff \theta = \epsilon \\
\theta, \mathbf{i} \models p * q & \iff \exists \theta_1, \theta_2.\, (\theta_1 * \theta_2)\!\downarrow \wedge\, \theta_1 * \theta_2 = \theta \wedge (\theta_1, \mathbf{i} \models p) \wedge (\theta_2, \mathbf{i} \models q) \\
\theta, \mathbf{i} \models p \mathrel{-\!\!*} q & \iff \forall \theta'.\, ((\theta * \theta')\!\downarrow \wedge\, (\theta', \mathbf{i} \models p)) \Rightarrow (\theta * \theta', \mathbf{i} \models q)
\end{array}$$

With the aid of $\mathrel{-\!\!*}$, called separating implication, we can define $p(X)$ for an assertion p as syntactic sugar for

$$\mathsf{true} * (\mathsf{emp} \wedge (X \mathrel{-\!\!*} p)).$$

Informally, $p(X)$ does not restrict the current state, but requires that X be bound to some state satisfying p. We use it in our extension of the logic for reasoning about linearizability (Appendix D). The above assertion language can be extended as needed when we consider particular instantiations of Σ. For all assertion languages we introduce in this paper, we use the usual operator $[\![\cdot]\!]$ for computing assertion denotations. For instance, for the language above, $[\![p]\!] = \{(\theta, \mathbf{i}) \mid (\theta, \mathbf{i} \models p)\}$.

We also assume a language for denoting parameterised predicates. For an assertion p denoting a parameterised predicate and $t \in \mathsf{ThreadID}$, the assertion p_t denotes the predicate $[\![p]\!]_t$.

Since RGSep partitions the program state into thread-local and shared parts, it has to extend the above assertion language so that assertions denote subsets of $\Sigma \times \Sigma \times \mathsf{LInt}$. Here the first component represents the state *local* to the thread in whose code the assertion is located, and the second, the *shared* state. The assertion language of RGSep is as follows:

$$P, Q ::= p \mid \boxed{p} \mid \mathsf{true} \mid \mathsf{false} \mid \exists X.\, P \mid \forall X.\, P \mid P * Q \mid P \wedge Q \mid P \vee Q$$

with the following semantics:

$$\begin{array}{ll} \theta, \theta', \mathbf{i} \models p & \iff \theta, \mathbf{i} \models p \\ \theta, \theta', \mathbf{i} \models \boxed{p} & \iff \theta = \epsilon \wedge (\theta', \mathbf{i} \models p) \\ \theta, \theta', \mathbf{i} \models P * Q & \iff \exists \theta_1, \theta_2.\, \theta = \theta_1 * \theta_2 \wedge (\theta_1, \theta', \mathbf{i} \models P) \wedge (\theta_2, \theta', \mathbf{i} \models Q) \end{array}$$

An assertion p denotes the local-shared state pairs with the local state satisfying p; $\boxed{p}$ the pairs with the empty local state and the shared state satisfying p; $P * Q$ the pairs in which the local state can be divided into two substates such that one of them together with the shared state satisfies P and the other together with the shared state satisfies Q. The semantics of $\wedge$ and $\vee$ is standard.

The judgements of our logic include rely and guarantee conditions determining how a command or its environment changes the shared state. To this end, we assume a language for expressing such conditions $R, G, \ldots$, denoting relations over $\Sigma \times \Sigma$. The conditions are often expressed using *actions* of the form $p \rightsquigarrow q$. Informally, this action changes the part of the shared state that satisfies p into one that satisfies q, while leaving the rest of the shared state unchanged. Formally, its meaning is the following relation on shared states:

$$[\![p \rightsquigarrow q]\!] = \{(\theta_1 * \theta_0, \theta_2 * \theta_0) \mid \exists \mathbf{i}.\, (\theta_1, \mathbf{i}) \in [\![p]\!] \wedge (\theta_2, \mathbf{i}) \in [\![q]\!]\}.$$

It relates some initial state θ_1 satisfying the precondition p to a final state θ_2 satisfying the postcondition q. In addition, there may be some disjoint state θ_0 that is not affected by the action.

In the following we denote by $\mathsf{atomic}\,\{\, C \,\}$ a block of code C considered as one atomic primitive command. We allow nested atomic blocks.

The judgements of the logic have the form

$$\Gamma \mid R, G \vdash_t^w \{P\}\, C\, \{Q\}.$$

Here R and G are rely and guarantee conditions, and $w \in \{\mathsf{in}, \mathsf{out}\}$ indicates whether the command C is inside an atomic block or not. The symbol $t \in \mathsf{ThreadID}$ is a thread identifier, C is a command in the code of thread t, and P and Q are assertions describing the local state of the thread and the shared state. The context Γ is a syntactic version of a method specification for libraries with unspecified implementation, which we introduced in Section 6, where pre- and postconditions are syntactic assertions denoting parameterised predicates. We require that these assertions p be ***insensitive to logical variables*** in the following sense:

$$\forall t \in \mathsf{ThreadID}.\, \forall (\theta, \mathbf{i}) \in [\![p_t]\!].\, \forall \mathbf{i}'.\, (\theta, \mathbf{i}') \in [\![p_t]\!].$$

We also require that they be precise, i.e., that for any $\mathbf{i} \in \mathsf{LInt}$ and $t \in \mathsf{ThreadID}$ the predicate $\{\theta \mid (\theta, \mathbf{i}) \in [\![p_t]\!]\}$ be precise (Section 6).

Informally, our judgement $\Gamma \mid R, G \vdash_t^w \{P\}\, C\, \{Q\}$ assumes that the command C is run by thread t, its environment changes the shared state according to R, its initial state satisfies P, and the methods it calls satisfy the contracts in Γ. Given this, the judgement guarantees that the command is safe, changes the shared state according to G, and its final state (if it terminates) satisfies Q. We have a similar judgement

$$\Gamma \vdash \{P\}\, \mathcal{C}\, \{Q\}$$

for a client $\mathcal{C}$, which may call library methods specified in Γ.

We partition all the atomic commands in the program into those that access only the local state of the thread executing them and those that can additionally access the shared state. By convention, the latter commands are atomic blocks.

The proof rules of RGSep are summarised in Figure 5. Most of the rules are standard ones from Hoare logic. We have a single axiom for primitive commands executing on the thread-local state (PRIM), which allows any pre- and postconditions consistent with the semantics of the command. The axiom uses the following lifting of the denotations of primitive commands $c \in \mathsf{PComm}$ (Section 2) to $\Sigma \times \mathsf{LInt}$:

$$f_c^t(\theta, \mathbf{i}) = f_c^t(\theta) \times \{\mathbf{i}\}, \tag{18}$$

if $f_c^t(\theta) \neq \top$; $f_c^t(\theta, \mathbf{i}) = \top$, otherwise. When particular Σ and f_c^t are chosen, the axiom can be specialised to several syntactic versions, obtaining a concrete instance of the abstract logic presented here.

Commands accessing the shared state are handled by three rules—ATOMICOUT, ATOMICOUT-R and ATOMICIN. The ATOMICOUT rule assumes the environment does not interfere, i.e., the rely is empty. It combines the local state p of the current thread with the part of the shared state satisfying p_0, and runs C as if this combination were in the thread's local state. The rule then splits the resulting state into local and shared parts, determining the shared part as the one that satisfies the annotation q_0. The rule requires that the change C makes to the shared state be allowed by its guarantee G.

In reasoning about how a command changes the shared state, we need to make sure that its views of the state is up-to-date with whatever changes its environment could make. For this reason, the rule ATOMICOUT-R requires pre- and postconditions to be ***stable*** under the rely R, i.e., insensitive to changes allowed by the relation. Formally, an assertion P is stable under a rely R when

$$\forall(\theta, \theta_1, \mathbf{i}) \in [\![P]\!]. \forall \theta_2. (\theta_1, \theta_2) \in [\![R]\!] \Rightarrow (\theta, \theta_2, \mathbf{i}) \in [\![P]\!].$$

After checking stability, ATOMICOUT-R replaces the rely with the empty set, thus enabling the application of ATOMICOUT.

Finally, the ATOMICIN rule just ignores nested atomic blocks.

The consequence rule (CONSEQ) allows strengthening the precondition and the rely and weakening the postcondition and the guarantee. The disjunction rule (DISJ) is useful for proof by cases. EXISTS1 rule is a usual rule from Hoare logic, and EXISTS2, its generalisation to logical variables ranging over states. The frame rule (FRAME) ensures that if a command C is safe when run from states in P, it does not touch an extra piece of state described by F. We have to require that the frame F be stable under both the rely R and the guarantee G, in case C contains commands changing the shared state.

The CALL axiom is a variation on the procedure call axiom from Hoare logic, handling calls to methods in Γ with unspecified implementations. To prove a call to a method m with a specification $\{p\}\, m\, \{q\}$, we instantiate p and q with the current thread identifier t, dispose the method precondition p_t from the pre-state, and allocate the method postcondition q_t to the post-state. The disposal and allocation here model the ownership transfer between the library of m and its client. Note that p and q restrict only the thread-local state of thread t.

The PAR rule combines judgements about several threads. Note that every thread in the rule assumes that the others satisfy their respective guarantees. Pre- and postconditions of threads in the premisses of the rule are $*$-conjoined in the conclusion.

$$\frac{f_c^t([\![p]\!]) \sqsubseteq [\![q]\!]}{\Gamma \mid R, G \vdash_t^w \{p\}\, c\, \{q\}}\ \textsc{Prim}$$

$$\frac{\Gamma \mid \emptyset, \emptyset \vdash_t^{\mathsf{in}} \{p * p_0\}\, C\, \{q * q_0\} \qquad [\![p_0 \rightsquigarrow q_0]\!] \subseteq G}{\Gamma \mid \emptyset, G \vdash_t^{\mathsf{out}} \{p * \boxed{p_0 * r}\}\ \mathsf{atomic}\,\{\, C \,\}\, \{q * \boxed{q_0 * r}\}}\ \textsc{AtomicOut}$$

$$\frac{\Gamma \mid \emptyset, G \vdash_t^{\mathsf{out}} \{P\}\, C\, \{Q\} \qquad P, Q \text{ stable under } R}{\Gamma \mid R, G \vdash_t^{\mathsf{out}} \{P\}\ \mathsf{atomic}\,\{\, C \,\}\, \{Q\}}\ \textsc{AtomicOut-R}$$

$$\frac{\Gamma \mid R, G \vdash_t^{\mathsf{in}} \{p\}\, C\, \{q\}}{\Gamma \mid R, G \vdash_t^{\mathsf{in}} \{p\}\ \mathsf{atomic}\,\{\, C \,\}\, \{q\}}\ \textsc{AtomicIn}$$

$$\frac{\Gamma \mid R, G \vdash_t^w \{P_1\}\, C_1\, \{P_2\} \qquad \Gamma \mid R, G \vdash_t^w \{P_2\}\, C_2\, \{P_3\}}{\Gamma \mid R, G \vdash_t^w \{P_1\}\, C_1; C_2\, \{P_3\}}\ \textsc{Seq}$$

$$\frac{\Gamma \mid R, G \vdash_t^w \{P\}\, C_1\, \{Q\} \qquad \Gamma \mid R, G \vdash_t^w \{P\}\, C_2\, \{Q\}}{\Gamma \mid R, G \vdash_t^w \{P\}\, C_1 + C_2\, \{Q\}}\ \textsc{Choice}$$

$$\frac{\Gamma \mid R, G \vdash_t^w \{P\}\, C\, \{P\}}{\Gamma \mid R, G \vdash_t^w \{P\}\, C^*\, \{P\}}\ \textsc{Loop}$$

$$\frac{P_1 \Rightarrow P_2 \quad R_1 \Rightarrow R_2 \quad \Gamma \mid R_2, G_2 \vdash_t^w \{P_2\}\, C\, \{Q_2\} \quad G_2 \Rightarrow G_1 \quad Q_2 \Rightarrow Q_1}{\Gamma \mid R_1, G_1 \vdash_t^w \{P_1\}\, C\, \{Q_1\}}\ \textsc{Conseq}$$

$$\frac{\Gamma \mid R, G \vdash_t^w \{P_1\}\, C\, \{Q_1\} \qquad \Gamma \mid R, G \vdash_t^w \{P_2\}\, C\, \{Q_2\}}{\Gamma \mid R, G \vdash_t^w \{P_1 \vee P_2\}\, C\, \{Q_1 \vee Q_2\}}\ \textsc{Disj}$$

$$\frac{\Gamma \mid R, G \vdash_t^w \{P\}\, C\, \{Q\}}{\Gamma \mid R, G \vdash_t^w \{\exists x.\, P\}\, C\, \{\exists x.\, Q\}}\ \textsc{Exists1}$$

$$\frac{\Gamma \mid R, G \vdash_t^w \{P\}\, C\, \{Q\}}{\Gamma \mid R, G \vdash_t^w \{\exists X.\, P\}\, C\, \{\exists X.\, Q\}}\ \textsc{Exists2}$$

$$\frac{\Gamma \mid R, G \vdash_t^w \{P\}\, C\, \{Q\} \qquad F \text{ is stable under } R \cup G}{\Gamma \mid R, G \vdash_t^w \{P * F\}\, C\, \{Q * F\}}\ \textsc{Frame}$$

$$\frac{}{\Gamma, \{p\}\, m\, \{q\} \mid R, G \vdash_t^w \{P * p_t\}\, m\, \{P * q_t\}}\ \textsc{Call}$$

$$\frac{\Gamma \mid R_1, G_1 \vdash_1^w \{P_1\}\, C_1\, \{Q_1\} \quad \ldots \quad \Gamma \mid R_n, G_n \vdash_n^w \{P_n\}\, C_n\, \{Q_n\}}{\Gamma \vdash \{P_1 * \ldots * P_n\}\, C_1 \parallel \ldots \parallel C_n\, \{Q_1 * \ldots * Q_n\}}\ \textsc{Par}$$

(where $R_t = \bigcup\{G_k \mid 1 \le k \le n \wedge k \neq t\}$)

Fig. 5. Proof rules of RGSep

According to the semantics of the assertion language, this takes the disjoint composition of the local states of the threads and enforces that the threads have the same view of the shared state.

The following theorems show how the logic can be used to establish the safety of open programs with and without a ground client.

Theorem 36 (Soundness—client). *Consider a program $\Gamma \vdash \mathcal{C}$ with a set of initial states I. Assume an assertion P is such that*

$$\forall \theta \in I.\, \exists (\theta_1, \theta_2, \mathbf{i}) \in [\![P]\!].\, \theta = \theta_1 * \theta_2.$$

If $\Gamma \vdash \{P\}\, \mathcal{C}\, \{Q\}$, then $\mathcal{C}$ is safe for I.

Theorem 37 (Soundness—library). *Consider a program $\mathcal{L} : \Gamma$ with a set of initial states I. Assume a guarantee G and an assertion* inv *insensitive to logical variables such that*

- $\forall \theta \in I.\, \exists (\theta, \mathbf{i}) \in [\![\mathsf{inv}]\!]$;
- $\boxed{\mathsf{inv}}$ *is stable under G;*
- *for all $\{p^m\}\, m\, \{q^m\} \in \Gamma$ and $t \in \mathsf{ThreadID}$*

$$\emptyset \mid G, G \vdash_t^w \{\boxed{\mathsf{inv}} * p_t^m\}\ C_m\ \{\boxed{\mathsf{inv}} * q_t^m\}.$$

Then $\mathcal{L}$ is safe for I.

The proofs are identical to the soundness proof of RGSep [16].

D Logic for Linearizability with Ownership Transfer

We now extend the logic presented in Appendix C to reason about our notion of linearizability (Section 3). The logic we present here generalises the method of proving linearizability using linearization points [1,12,16] to the setting with ownership transfer.

Consider the following program $\vdash \mathcal{L} : \Gamma$ with a set of initial states I:

$$\mathcal{L} = \mathsf{let}\ \{m = C_m \mid m \in M\}\ \mathsf{in}\ [-];$$
$$\Gamma = \{\{p^m\}\, m\, \{q^m\} \mid m \in M\}.$$

The method of linearization points is restricted to proving the linearization of $\mathcal{L}$ by a library $\mathcal{L}'$ with all methods implemented atomically. Our goal is thus to establish that $(\mathcal{L}, I) \sqsubseteq (\mathcal{L}', I')$, where

$$\mathcal{L}' = \mathsf{let}\ \{m = (\mathsf{skip}^*; \mathsf{atomic}\, \{\, C_m^a \,\}; \mathsf{skip}^*) \mid m \in M\}\ \mathsf{in}\ [-].$$

Proof systems for linearizability typically do not allow library specification to make any statements about liveness properties of the library. Thus, the skip^* statements in the abstract library implementation $\mathcal{L}'$ allow for any termination behaviour of its methods: the first one models the divergence before the method makes a change to the library state using C_m^a, and the second, the divergence after this. Here we restrict ourselves to verifying linearizability with respect to abstract implementations capturing only safety properties of a library; liveness properties can be handled following [10].

The method of linearization points considers the concrete and the abstract implementations of the library running alongside each other. Both are run under their most

general clients; however, while we consider all possible executions of the client of the concrete library, the client of the abstract one is allowed to call a method only when the corresponding concrete method implementation is at a certain ***linearization point***. The linearization point thus determines the place where the concrete implementation of the method 'takes effect'. Proving linearizability then boils down to checking that, if the abstract method invocation receives the ownership of the same piece of state upon its call as the corresponding concrete one, then they will both return the same state at their returns. The sequence of abstract method invocations at linearization points in an execution of the concrete implementation yields the desired linearizing history of the abstract implementation. The conditions restricting possible rearrangements of method invocations in the definition of linearizability are trivially satisfied, since a linearization point is inside the code of the corresponding concrete method.

The above method typically requires relating the states of the two library implementations. For this reason, we adjust the assertion language of Appendix C to describe such relations. Namely, we define a new syntactic category

$$p_r, q_r ::= \mathsf{true} \mid \mathsf{false} \mid X \mid \exists X.\, p_r \mid \forall X.\, p_r \mid p_r \wedge q_r \mid \\ p_r \vee q_r \mid \mathsf{emp} \mid p_r * q_r \mid p \mid \lfloor p \rfloor$$

of relational assertions, where $p, q, \ldots$ range over assertions denoting subsets of $\Sigma \times \mathsf{LInt}$. The assertion $\lfloor p \rfloor$ denotes a state of the abstract implementation satisfying p. Note that we disallow nested $\lfloor \cdot \rfloor$ operators.

The assertions p_r, q_r denote subsets of $\Sigma \times \Sigma \times \mathsf{LInt}$ according to the following semantics:

$$\theta_c, \theta_a, \mathbf{i} \models p \iff (\theta_c, \mathbf{i} \models p) \wedge \theta_a = \epsilon$$
$$\theta_c, \theta_a, \mathbf{i} \models \lfloor p \rfloor \iff (\theta_a, \mathbf{i} \models p) \wedge \theta_c = \epsilon$$

We then modify the assertion language of RGSep as follows:

$$P_r ::= \ldots \mid p_r \mid \boxed{p_r} \mid \mathsf{Pre}(p) \mid \mathsf{Post}(p)$$

This allows pieces of abstract state to be local or shared. The Pre and Post assertions are used to reason about the correspondence between pre- and postconditions of the concrete and the abstract library implementations (see below). The resulting assertions denote subsets of

$$\Sigma^2 \times \Sigma^2 \times \Sigma \times \{\mathsf{Pre}, \mathsf{Post}, \mathsf{None}\} \times \mathsf{LInt}.$$

Here the first component represents a pair of thread-local states of the concrete and abstract implementations, and the second, a pair of shared states of these two implementations. The next two components record the state given in a Pre or a Post assertion. The semantics of the new assertion language is given in Figure 6. The definitions for assertions not in the figure are obtained from the corresponding cases in the logic of Appendix C either by ignoring the components corresponding to Pre and Post, like in the case of p_r, or by propagating them to sub-assertions, like in the case of $P_r \wedge Q_r$.

Finally, we assume a language for expressing rely-guarantee conditions $R_r, G_r, \ldots$ over pairs of concrete and abstract shared states; thus, R_r denotes a subset of $\Sigma^2 \times \Sigma^2$.

The judgements of the new proof system have the form

$$\Gamma \mid R_r, G_r \vdash_t^{w,j} \{P_r\}\ C\ \{Q_r\},$$

$$
\begin{array}{l}
(\theta_c, \theta_a), (\theta'_c, \theta'_a), \theta, \nu, \mathbf{i} \models \mathsf{Pre}(p) \iff \theta, \mathbf{i} \models p \wedge \nu = \mathsf{Pre} \\
(\theta_c, \theta_a), (\theta'_c, \theta'_a), \theta, \nu, \mathbf{i} \models \mathsf{Post}(p) \iff \theta, \mathbf{i} \models p \wedge \nu = \mathsf{Post} \\
(\theta_c, \theta_a), (\theta'_c, \theta'_a), \theta, \nu, \mathbf{i} \models P_r * Q_r \iff \\
\quad \exists \theta^1_c, \theta^2_c, \theta^1_a, \theta^2_a, \theta_1, \theta_2, \nu_1, \nu_2.\, \theta_c = \theta^1_c * \theta^2_c \wedge \theta_a = \theta^1_a * \theta^2_a \wedge \\
\quad ((\theta^1_c, \theta^1_a), (\theta'_c, \theta'_a), \theta_1, \nu_1, \mathbf{i} \models P_r) \wedge ((\theta^2_c, \theta^2_a), (\theta'_c, \theta'_a), \theta_2, \nu_2, \mathbf{i} \models Q_r) \wedge \\
\quad ((\nu_1 = \mathsf{None} \wedge \nu_2 = \mathsf{None} \wedge \nu = \mathsf{None}) \vee \\
\quad\ (\nu_1 = \mathsf{None} \wedge \nu_2 \in \{\mathsf{Pre}, \mathsf{Post}\} \wedge \nu = \nu_2 \wedge \theta_2 = \theta) \vee \\
\quad\ (\nu_1 \in \{\mathsf{Pre}, \mathsf{Post}\} \wedge \nu_2 = \mathsf{None} \wedge \nu = \nu_1 \wedge \theta = \theta_1)) \\
(\theta_c, \theta_a), (\theta'_c, \theta'_a), \theta, \nu, \mathbf{i} \models p_r \iff \theta_c, \theta_a, \mathbf{i} \models p_r \\
(\theta_c, \theta_a), (\theta'_c, \theta'_a), \theta, \nu, \mathbf{i} \models \boxed{p_r} \iff \theta'_c, \theta'_a, \mathbf{i} \models p_r \wedge \theta_c = \theta_a = \epsilon \\
(\theta_c, \theta_a), (\theta'_c, \theta'_a), \theta, \nu, \mathbf{i} \models P_r \wedge Q_r \iff \\
\quad ((\theta_c, \theta_a), (\theta'_c, \theta'_a), \theta, \nu, \mathbf{i} \models P_r) \wedge ((\theta_c, \theta_a), (\theta'_c, \theta'_a), \theta, \nu, \mathbf{i} \models Q_r)
\end{array}
$$

Fig. 6. Semantics of the assertion language of the logic for linearizability

where j is conc or abs, depending on whether C is a command belonging to the concrete or the abstract implementation.

Having changed the assertion language, we now have to adjust some of the proof rules in Figure 5. First, we lift the transformers f^t_c defining the semantics of primitive commands c to $\Sigma \times \Sigma$ in the following two ways: for all $\theta, \theta' \in \Sigma$

$$
\begin{array}{l}
f^t_{\mathsf{conc}(c)}(\theta, \theta') = f^t_c(\theta) \times \{\theta'\}; \\
f^t_{\mathsf{abs}(c)}(\theta, \theta') = \{\theta\} \times f^t_c(\theta'),
\end{array}
$$

if the value of f^t_c on the corresponding single state is not $\top$; and $\top$, otherwise. The transformers define the effect of c executing on the concrete, respectively, the abstract state. We then lift $f^t_{j(c)}$ to $\Sigma \times \Sigma \times \mathsf{LInt}$ similarly to (18). We now replace the PRIM axiom with two corresponding variants:

$$
\frac{f^t_{j(c)}([\![p_r]\!]) \sqsubseteq [\![q_r]\!]}{\Gamma \mid R, G \vdash^{w,j}_t \{p_r\}\ c\ \{q_r\}}\ \textsc{Prim}\text{-}j
$$

for $j \in \{\mathsf{conc}, \mathsf{abs}\}$. The rest of the rules of the new logic are obtained from the rules in Figure 5 by replacing $\vdash_t$ with $\vdash^j_t$. We only need to change the notion of stability: P_r is stable under R_r when

$$
\begin{array}{l}
\forall((\theta_c, \theta_a), (\theta^c_1, \theta^a_1), \theta, \nu, \mathbf{i}) \in [\![P_r]\!].\, \forall \theta^c_2, \theta^a_2 \in \Sigma. \\
\qquad\qquad ((\theta^c_1, \theta^a_1), (\theta^c_2, \theta^a_2)) \in [\![R_r]\!] \Rightarrow ((\theta_c, \theta_a), (\theta^c_2, \theta^a_2), \theta, \nu, \mathbf{i}) \in [\![P_r]\!].
\end{array}
$$

The definition of $p_r \rightsquigarrow q_r$ is adjusted similarly.

For an implementation C_m of a method m in the domain of Γ, let $\tilde{C}_m$ be C_m with some of commands C inside atomic blocks replaced by $C; (\!|C^a_m|\!)$. Here we mark with $(\!|\cdot|\!)$ the code of the abstract implementation, which is treated specially by our proof system. The placement of $(\!|C^a_m|\!)$ thus fixes linearization points inside the code of C_m.

The proof method for linearizability builds on the technique for proving the safety of library implementations described in Theorem 37. To prove $(\mathcal{L}, I) \sqsubseteq (\mathcal{L}', I')$, we require a guarantee G_r and an assertion inv_r insensitive to logical variables such that

– $\forall \theta \in I.\, \exists \theta' \in I'.\, \exists \mathbf{i}.\, (\theta, \theta', \mathbf{i}) \in [\![\mathsf{inv}_r]\!]$; and

– $\boxed{\mathsf{inv}_r}$ is stable under G_r.

For every thread t and $\{p^m\}\ m\ \{q^m\} \in \Gamma$, we require a derivation of the following judgement:

$$\emptyset \mid G_r, G_r \vdash_t^{\mathsf{conc}} \{\exists X.\, \mathsf{Pre}(X) \wedge p_t^m(X) \wedge X * \boxed{\mathsf{inv}}\}\ \tilde{C}_m\ \{\exists Y.\, \mathsf{Post}(Y) \wedge q_t^m(Y) \wedge Y * \boxed{\mathsf{inv}}\} \tag{19}$$

with abstract method implementations $(\!|C_m^a|\!)$ inside C_m handled using the following proof rule:

$$\frac{\emptyset \mid R_r', G_r' \vdash_t^{\mathsf{in,abs}} \{\exists X.\, p_t^m(X) \wedge \lfloor X \rfloor * P_r)\}\ C_m^a\ \{\exists Y.\, q_t^m(Y) \wedge \lfloor Y \rfloor * Q_r\}}{\emptyset \mid R_r', G_r' \vdash_t^{\mathsf{in,conc}} \{\exists X.\, \mathsf{Pre}(X) \wedge P_r\}\ (\!|C_m^a|\!)\ \{\exists Y.\, \mathsf{Post}(Y) \wedge Q_r\}}\ \textsc{LinPoint}$$

The rationale behind (19) and LINPOINT is as follows. As noted above, in our proof method we assume that the abstract implementation, when executed at a linearization point, receives the ownership of the same piece of state as the concrete one called earlier. We then have to establish that the piece of state the abstract implementation returns to the client at the linearization point is the same as what the concrete one returns at the method return. Pre and Post predicates are used to reason about such relationships. In the precondition of (19), $\mathsf{Pre}(X)$ records the state X received by the concrete implementation when it was called, which is assumed to satisfy the precondition p_t^m. According to the precondition of the premiss of LINPOINT, the abstract implementation then receives the abstract copy $\lfloor X \rfloor$ of the state passed to the concrete implementation at its invocation, as recorded by the $\mathsf{Pre}(X)$ predicate. Here we can also assume that the state X satisfies the precondition p_t^m. LINPOINT requires the abstract implementation C_m^a to be verified in the abstract proof system, where primitive commands can only act on the abstract state. In the postcondition of the premiss of LINPOINT, we require the abstract implementation to produce a piece of abstract state $\lfloor Y \rfloor$ such that Y satisfies q_t^m. The postcondition of the conclusion of LINPOINT then records this state in $\mathsf{Post}(Y)$. Finally, $\mathsf{Post}(Y)$ is used in the postcondition of (19) to check that the concrete implementation returns the same state as the abstract one. All proof rules except LINPOINT do not change Pre and Post, treating them as ghost state.

The proof method also requires the abstract implementation to be executed at most once during the execution of a concrete one. To this end, LINPOINT rule exchanges a Pre assertion for a Post one, and the semantics of $*$ prohibits duplicating the assertions. This ensures that only one linearization point can be present in any execution of the method. Note that, in LINPOINT, P_r can contain free occurrences of X and Q_r of Y, thus correlating the current state with the auxiliary information in Pre and Post predicates.

Theorem 38 (Soundness). *Under the above conditions, $\mathcal{L}$ and $\mathcal{L}'$ are safe for I and I', respectively, and $(\mathcal{L}, I) \sqsubseteq (\mathcal{L}', I')$.*

Despite the presented logic being based on an existing method of linearization points, our extension to ownership transfer is new, with the main technical novelty being our use of logical variables ranging over states to track correlations between concrete and abstract pre- and postconditions.