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* [9]: 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 [5] 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, 6, 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 [10, Appendix B].

2 Footprints of States

Our results hold for a class of models of program states called *separation algebras* [4], 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 [10, 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 [10, 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 $\mathsf{ThreadID}$ 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:

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

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* $\mathsf{BHistory}$ *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 [8]. 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 [10, 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 [10, 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:

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

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 $\lambda m, t.\, (\!|C_m|\!)^{-}_{t}(\_)$ 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

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

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

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

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

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

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*

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

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 [9]. 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 $\mathsf{client}(\tau_2) = \mathsf{ground}(\eta) = \mathsf{client}(\tau_1)$. $\square$

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 [10, 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 $\lfloor\!\lfloor H \rfloor\!\rfloor_\Gamma$ be the result of replacing every action φ in H by the action $\lfloor\!\lfloor \varphi \rfloor\!\rfloor_\Gamma$ defined as follows:

$$\lfloor\!\lfloor (t, \mathsf{call}\ m(\theta)) \rfloor\!\rfloor_\Gamma = (t, \mathsf{call}\ m(\theta \setminus p_t^m)); \quad \lfloor\!\lfloor (t, \mathsf{ret}\ m(\theta)) \rfloor\!\rfloor_\Gamma = (t, \mathsf{ret}\ m(\theta \setminus q_t^m)).$$

$\lfloor\!\lfloor H \rfloor\!\rfloor_\Gamma$ 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 [9] and weak memory models [3]. 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 [3], 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 [1,6,16]. 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. [6,7] 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 [2,8]. 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. S. Burckhardt, A. Gotsman, M. Musuvathi, and H. Yang. Concurrent library correctness on the TSO memory model. In *ESOP*, 2012.
4. C. Calcagno, P. O'Hearn, and H. Yang. Local action and abstract separation logic. In *LICS*, 2007.
5. D. G. Clarke, J. Noble, and J. M. Potter. Simple ownership types for object containment. In *ECOOP*, 2001.
6. T. Elmas, S. Qadeer, A. Sezgin, O. Subasi, and S. Tasiran. Simplifying linearizability proofs with reduction and abstraction. In *TACAS*, 2010.
7. T. Elmas, S. Qadeer, and S. Tasiran. A calculus of atomic actions. In *POPL*, 2009.
8. 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.
9. A. Gotsman and H. Yang. Liveness-preserving atomicity abstraction. In *ICALP*, 2011.
10. Alexey Gotsman and Hongseok Yang. Linearizability with ownership transfer (extended version). Available from `www.software.imdea.org/~gotsman`, 2012.
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.