The Importance of Being Linearizable

Abstract

Linearizability is a commonly accepted notion of correctness for libraries of concurrent algorithms. Even though most algorithms getting published are shown to satisfy it, so far the notion has not been defined for realistic settings where they actually get used. Furthermore, it has not been clear if in such settings the linearizability of a library tells us anything about the behaviour of its clients. Rather, linearizability has mostly served as a box to be ticked in a paper with a new concurrent algorithm, lest it should get rejected.

In this paper, we show that linearizability is more important than that. First, we generalise it to cope with a realistic setting, including multiple libraries and clients executing in a shared address space. We do not limit possible interactions between them to passing values of a primitive data type, but allow transferring the ownership of memory areas. Second, we show that, despite such subtle interactions, the proposed notion of linearizability allows decomposing the verification of a whole program into the verification of its constituent components: 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. This decomposition is of a finer grain than the one enabled by thread-modular methods. We demonstrate the use of our method by modularising the verification of two challenging concurrent algorithms.

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 Intel's 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 the notion of *linearizability* [16], which fixes a certain correspondence between the library and its abstract specification (the latter usually sequential, with methods implemented atomically). Unfortunately, this notion is not appropriate for the settings where concurrent libraries actually get used. In particular, the classical definition of linearizability assumes a complete isolation between a library and its client, with interactions limited to passing elements of a primitive data type as parameters or return values of library methods. In reality, 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, 7] and program logics [24] usually achieve this using the concept of *ownership* of data structures by a program component: the right to access a data structure is given only to a particular component or a set of them. In real software, this right is not assigned statically, and the ownership of data structures can be *transferred* between components at certain points, such as calls to and returns from a library. Such interactions have to be taken into account when defining linearizability in a realistic setting.

Examples. For an example of ownership transfer between concurrent libraries and their clients, 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, in a programmer's mind (and in a formal proof of correctness), the container usually also holds the ownership of the data structures whose addresses it stores. 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 will result in a race condition. Hence, to be useful in practice, linearizability of a concurrent container cannot be defined only in terms of passing pointers between the container and the client, as has been the case so far [12, 14, 16]; it must ensure that the container correctly transfers the ownership of data structures associated with the pointers.

Figure 1. A sketch of a concurrent queue implementation

```
struct Node;

void enqueue(int x) {
  ...
  Node *newnode =
    alloc(sizeof(Node));
  ...
  // linearization point
  ...
}

int isEmpty() { ... }
```

```
int dequeue() {
  ...
  // linearization point
  ...
  free(oldnode);
  ...
  // Wrapper *ret =
  // alloc(sizeof(Wrapper));
  ...
  // return ret;
  return value;
}
```

For another example of ownership transfer, 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. When a thread allocates a memory block, it gets the exclusive ownership of the block, which allows it to access the block without interference from the other threads. When the thread frees the block, the ownership is returned to the allocator. Trying to write to a memory cell after it was freed has dire consequences.

Speaking of which, a memory allocator is used by almost every concurrent library. However, the standard notion of linearizability does not capture the effects that the interactions between the library and the allocator might have on the client. The most obvious of such effects is that the library using a lot of memory might cause allocations in the client to fail. However, there are more subtle effects highlighting a fundamental problem not specific to memory allocation. Consider a sketch of a concurrent queue implementation in Figure 1 (a full implementation is given in Appendix E). The queue stores integers and provides three methods with self-explanatory names. There are a number of such implementations, supposedly linearizable with respect to the abstract queue data type with the three operations implemented atomically [21]. Their linearizability is usually established by identifying *linearization points* in the code of every implemented method at which, informally, the method "takes effect"—the change it makes to the library state becomes visible to the other threads (see [16] and Appendix D). The `enqueue` method allocates a `Node` structure to represent a new element inside the queue; `dequeue` deallocates it when the element is removed.

Now, let us modify the library such that, instead of returning the value dequeued directly, the `dequeue` method returns it in a wrapper object:

```
struct Wrapper *dequeue();
```

This requires one more call to the memory allocator to allocate the wrapper; the implementation of `dequeue` is changed accordingly by uncommenting the extra lines in its body. As it happens, the additional call to the allocator to get a new wrapper makes the library non-linearizable. When a library is linearizable, the informal expectation (formalised in Section 5) is that the behaviour of its clients should be reproducible when they use the atomic abstract library implementation instead. Thus, consider the following client:

```
int flag = 0, res = 0;
Wrapper *x, *y;
enqueue(10);
in_parallel((y = dequeue()),
            (flag = isEmpty();
             x = alloc(sizeof(Wrapper));
             free(x)));
if (flag && y == x) res = 1;
```

This client can set `res` to 1 when using our queue implementation. Indeed, assume `dequeue` returns a wrapper allocated at address 42. Then during the time between removing the node storing 10 at the old linearization point of `dequeue` and allocating a wrapper using `alloc`, the queue is empty, but cell 42 is free. This can be noticed by the second thread of the client and lead to `res` getting assigned to 1. If the new library were linearizable, the final value 1 of `res` could be obtained with library methods performing queue operations and wrapper allocation atomically. This, however, cannot happen: in this case, if the queue is empty, then the wrapper to be returned by `dequeue` has already been allocated.

Hence, the modified queue is not linearizable, while, as we show later (Section 6 and Appendix E), the original implementation is. This problem is not specific to memory allocation and could manifest itself had the queue implementation called any other library accessible to the client. The trouble is that linearizability cannot cope with multiple libraries interacting with each other.

Contributions. As is demonstrated by the above discrepancies between the theory of linearizability and the practice of concurrent programming, even though linearizability has been accepted as a correctness condition for concurrent libraries, so far there has been no definition of the notion applicable to the environments in which libraries execute in practice! Furthermore, it has not been clear if in such environments the linearizability of a library tells us anything about the behaviour of its clients. Rather, linearizability has mostly served as a box to be ticked in a paper with a new concurrent algorithm, lest it should get rejected. In this paper, we show that linearizability is more important than that:

- We generalise it to cope with a realistic setting, including multiple libraries and clients executing in a shared address space, whose interactions involve ownership transfers (Section 4).
- We show that, despite such subtle interactions, the proposed notion of linearizability allows decomposing the verification of a whole program into the verification of its constituent components. Namely, we establish an Abstraction Theorem (Theorem 9, Section 5): 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. The abstract implementation is usually simpler than the original one (in most cases implemented atomically), which eases the proof of the resulting program.
- We also show that our Abstraction Theorem allows abstracting several non-recursive interacting libraries (Section 5.1). This extension is non-trivial, since, while abstracting one library, we have to preserve certain relationships between it and the other libraries required by linearizability. This calls for a stronger statement of Abstraction Theorem.
- Our formal development does not rely on a particular model of program states, but assumes an arbitrary model from a certain class (Section 2). By picking a model with so-called *permissions* [4, 8], we can allow clients and libraries to transfer non-exclusive rights to access certain memory areas in particular ways, instead of transferring their full ownership. This makes our Abstraction Theorem applicable even when libraries and their clients share access to some areas of memory.
- We demonstrate that the Abstraction Theorem is not just a theoretical result: it enables compositional reasoning about complex concurrent algorithms that are challenging for existing verification methods. Namely, we modularly verify the linearizability of a non-blocking queue [21] using a custom memory allocator implemented as a non-blocking stack [30], and a multiple-word compare-and-swap (MCAS) algorithm [15] implemented using an auxiliary operation. The examples exhibit both ownership transfer and non-disjointness of the library states. In both cases, the Abstraction Theorem makes the proof tractable by allowing us to verify the linearizability of one part of the algorithm (e.g., the queue implementation) assuming an atomic specification of the other part it uses (e.g., the allocator). To verify the examples, we developed a logic for establishing our notion of linearizability, based on separation logic [29]. Due to space constraints, we describe it in Appendix D. Here we instead explain the benefits the Abstraction Theorem gives us in the examples considered (Section 6).

Technical challenges. Generalising linearizability to the realistic setting described above and proving the corresponding Abstraction Theorem presents several challenges. First, 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: while generating histories of a library we should only consider its executions in an environment that respects the notion of ownership. For example, a client that transfers a piece of state upon a call to a library not communicating with anyone else cannot then transfer the same piece of state again before getting it back from the library upon a method return. We propose a notion of *well-balancedness* that characterises those histories that treat ownership transfer correctly and should be taken into account when defining linearizability. The proof of our Abstraction Theorem relies crucially on the library semantics including only well-balanced histories. We define a way to generate such histories from a library implementation using the *most general client* of the library, which performs all possible ownership transfers consistent with the library specification (Section 3). As we show, the safety of the most general client, which can be established in existing program logics (Appendix C), implies that the library does not access internals of its clients.

Second, the Abstraction Theorem holds only for certain *healthy* clients that do not access the internals of the library and provide the pieces of memory expected by the library at every call. As we argue in Section 7, the notions of client healthiness used in data refinement [3, 13, 22], a sequential counterpart of the problem we are solving in this paper, do not generalise to concurrent setting. We formulate a more flexible notion appropriate for our application (Section 3) and show that it can be established in existing program logics (Appendix C).

Finally, the proof of the Abstraction Theorem is highly non-trivial in the presence of ownership transfer. It requires transforming a trace of a client using a concrete library implementation into a trace generated when the client uses an abstract implementation instead. As we discuss in Section 5.2, this involves: decomposing the original trace into traces in certain client- and library-local semantics (Lemma 12); defining a sequence of transformations on the client trace that make it consistent with a given history of interactions with the abstract library (Lemma 13); and, finally, composing the transformed client trace with a library-local trace of the abstract implementation (Lemma 14). These steps are delicate: e.g., the decomposition requires maintaining a splitting of the shared address space into parts owned by the library and the client, despite ongoing ownership transfers; the transformation requires the rearrangements being performed on the trace to preserve the correctness of ownership transfers to and from the library, and relies crucially on the histories involved being well-balanced.

Related work. We discuss related work in detail in Section 7. Here we briefly position our work with respect to the most common approach of decomposing the verification of concurrent programs—*thread-modular* reasoning methods. These consider every thread in

the program in isolation under some assumptions on its environment [19, 27]. However, a single thread would usually make use of multiple program components. This work goes further by allowing a finer-grain *intrathread-modular* reasoning: separating the verification of a library and its client, the code from both of which may be executed by a single thread. Note that this approach is complementary to thread-modular reasoning, which can still be used to carry out the verification subtasks, such as establishing the linearizability of libraries and proving the healthiness of clients. In fact, the logic we use for establishing linearizability (Appendix D) is thread-modular. In Section 7 we revisit this comparison in the light of examples of using our verification method (Section 6).

2. Preliminaries

Programming language. We present our results for a simple language of heap-manipulating concurrent programs. Let PComm be the set of primitive commands, ranged over by c, and Method the set of method names, ranged over by m. We assume that the methods do not take arguments and do not return values (these can be passed via the heap using other mechanisms we provide). The syntax of the language is given below:

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

The commands include primitive commands c, method calls m, sequential composition $C; C'$, nondeterministic choice $C + C'$ and iteration C^*. The standard constructs, such as loops and conditionals, can be defined as syntactic sugar (Appendix A).

A program consists of multiple *libraries* L implementing methods and their *ground client* $C_1 \parallel \ldots \parallel C_n$, given by a parallel composition of threads. In the following we index threads in programs using the set of identifiers $\mathsf{ThreadID} = \mathbb{N}$. A library implementation L defines a set of methods and declares some of them as atomic, meaning that calls to such methods run without being interleaved with the execution of other threads. We allow library implementations to call methods from other libraries, as is the case in the examples mentioned in Section 1. However, we do not allow recursion crossing library boundaries, leaving it for future work (Section 8). Throughout the paper, we assume that every method called in the program is defined by some library in the scope of the call, that different libraries define disjoint sets of methods, and that implementations of atomic methods do not call non-atomic ones.

State model. We do not fix a model of states for our programming language. Rather, our results hold for a class of models called *separation algebras* [6] that allow expressing the dynamic memory partitioning between libraries and their 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.*

In the rest of the paper we assume a separation algebra State with the operation $*$. We think of elements of State 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*. Incompatible states usually make contradictory claims about the ownership of memory.

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. In the future we also write $\_$ for an expression whose value is irrelevant and implicitly existentially quantified.

An example of a separation algebra is the following set RAM, often used to give semantics to heap-manipulating programs:

$$\mathsf{Loc} = \mathbb{N}^+ \qquad \mathsf{Val} = \mathbb{Z} \qquad \mathsf{RAM} = \mathsf{Loc} \rightharpoonup_{fin} \mathsf{Val}$$

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

More complicated separation algebras do not split memory completely, instead allowing heap parts combined by $*$ to overlap. This is done by associating so-called *permissions* [4] with memory cells in the model, which do not give their exclusive ownership, but allow accessing them in a certain way. Types of permissions range from read sharing [4] to accessing memory in an arbitrary way consistent with a given specification [8]. We give an example of a separation algebra with permissions in Appendix A. Since we develop all our results for an arbitrary separation algebra State, by instantiating it appropriately we can handle cases when a library and its client share access to some memory areas. For example, this is the case in one of the algorithms we verify in Section 6.

In the following formal development we also make use of the notion of *footprints*, which, informally, describe the equivalence class of heaps with the same allocated memory or permissions. Let us lift $*$ to $\mathcal{P}(\mathsf{State})$ pointwise: for $p, q \in \mathcal{P}(\mathsf{State})$

$$p * q = \bigcup \{\theta_1 * \theta_2 \mid \theta_1 \in p, \theta_2 \in q, (\theta_1 * \theta_2)\downarrow\}. \qquad (1)$$

Using the lifted $*$ operator, we now present a novel definition characterising the separation algebras that admit a notion of footprints.

Definition 2. *For a separation algebra Σ, let us define $\delta : \Sigma \to \mathcal{P}(\Sigma)$ as follows: for $\theta \in \Sigma$*

$$\delta(\theta) = \{\theta' \mid \forall \theta''.\, (\theta' * \theta'')\downarrow \Leftrightarrow (\theta * \theta'')\downarrow\}.$$

*We say that Σ is an **algebra with footprints** when*

1. *For all $\theta_1, \theta_2 \in \Sigma$, if $\theta_1 * \theta_2$ is defined, $\delta(\theta_1) * \delta(\theta_2) = \delta(\theta_1 * \theta_2)$.*
2. *For all $\theta_1, \theta_2, \theta_1', \theta_2' \in \Sigma$, if $\theta_1 * \theta_2$ and $\theta_1' * \theta_2'$ are defined,*

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

The function δ in the definition computes the equivalence class of states having the same footprint as θ. In the case of the algebra RAM, for $\theta \in \mathsf{RAM}$ we have $\delta(\theta) = \{\theta' \mid \mathsf{dom}(\theta) = \mathsf{dom}(\theta')\}$. Thus, states with the same footprint contain the same memory cells. It is easy to check that the conditions in Definition 2 are satisfied. Definitions of δ for separation algebras with permissions are more complicated, taking into account not only memory cells present in the state, but also permissions for them (Appendix A). In the following, we assume that State is an algebra with footprints. Let $\mathsf{Foot} = \{\delta(\theta) \mid \theta \in \mathsf{State}\}$ be the set of footprints.

Property 1 in Definition 2 implies that for $l_1, l_2 \in \mathsf{Foot}$ we have $l_1 * l_2 \in \mathsf{Foot} \cup \{\emptyset\}$. Thus the composition of two footprints produces either another footprint or the empty set, the latter in the case when the footprints being combined are incompatible.

Property 2 lets us define a partial subtraction operation on equivalence classes in Foot. For $l_1, l_2 \in \mathsf{Foot}$, if there are $\theta_1, \theta_2, \theta$ such that $\theta_1 \in l_1, \theta_2 \in l_2$ and $\theta_1 = \theta_2 * \theta$, we denote with $l_1 \setminus l_2 \in \mathsf{Foot}$ the equivalence class $\delta(\theta)$. When such $\theta_1, \theta_2, \theta$ do not exist, $l_1 \setminus l_2$ is undefined. It is easy to show that the $\setminus$ operation is well-defined.

In our proofs we also rely on an additional property of $*$ and $\setminus$ (Appendix A), which we omit to conserve space.

We say that a footprint l_1 is ***smaller*** than a footprint l_2, written $l_1 \preceq l_2$, when $l_2 \setminus l_1$ is defined. For $\theta_1, \theta_2 \in \mathsf{State}$ we let $\theta_1 \preceq \theta_2$ when $\delta(\theta_1) \preceq \delta(\theta_2)$. Finally, for $p_1, p_2 \in \mathcal{P}(\mathsf{State})$ we let $p_1 \preceq p_2$ when $\forall \theta_1 \in p_1.\, \exists \theta_2 \in p_2.\, \theta_1 \preceq \theta_2$.

Semantics of primitive commands. Consider the set $\mathcal{P}(\mathsf{State})^\top$ of subsets of State with a special element $\top$ used to denote an error state, resulting, e.g., from dereferencing an invalid pointer. We lift the $*$ operator to $\mathcal{P}(\mathsf{State})^\top$ as in (1) and by defining $\top * p = p * \top = \top * \top = \top$ for all $p \in \mathcal{P}(\mathsf{State})$.

Our semantics assumes an interpretation of every primitive command $c \in \mathsf{PComm}$ as a transformer $f_c^t : \mathsf{State} \to \mathcal{P}(\mathsf{State})^\top$, which maps pre-states to states obtained when thread $t \in \mathsf{ThreadID}$ executes c from the pre-state. Our semantics executes primitive commands atomically. Note that by defining appropriate primitive commands we can execute any block of code without method calls atomically. We use atomic methods in the syntax of programs to handle the latter case. The fact that our transformers are parameterised by t allows atomic accesses to the 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. In particular, method parameters and return values in our language can be passed via special locations on the heap associated with the index of the thread calling the method.

For our results to hold, we need to place the following restrictions on the transformer f_c^t for every primitive command $c \in \mathsf{PComm}$ and thread $t \in \mathsf{ThreadID}$:

Strong Locality: for any $\theta_1, \theta_2 \in \mathsf{State}$

$$(\theta_1 * \theta_2)\downarrow \wedge f_c^t(\theta_1) \neq \top \Rightarrow f_c^t(\theta_1 * \theta_2) = f_c^t(\theta_1) * \{\theta_2\}. \quad (2)$$

Footprint Preservation: for any $\theta, \theta' \in \mathsf{State}$

$$\theta' \in f_c^t(\theta) \Rightarrow \delta(\theta') = \delta(\theta). \quad (3)$$

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. This property is a variation on the locality property that ensures the soundness of separation logic [6]. It is stronger than locality because, while the latter prohibits the command from changing the additional state, it allows the effect of the command to depend on it; see [13] for a discussion.

Footprint preservation prohibits primitive commands from allocating or deallocating memory. This does not pose a problem, since, as we argued in Section 1, in the context of linearizability allocators should be treated as libraries. As we show in the next section, we permit receiving the ownership of memory cells from a library.

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

3. Ownership-transfer semantics of open programs

To formulate our results, we have to give a semantics to parts of programs in the language of Section 2, such as libraries considered in isolation from their clients and clients considered in isolation from implementations of libraries they use. In this section, we give a semantics to such *open* programs, and, as its special case, to *closed* programs of Section 2. The novelty of the notion of open programs we propose is that we allow the programs to communicate with their environment via ownership transfers. The semantics of libraries defined in this section is later used to formulate linearizability (Section 4), and the semantics of clients, the notion of client healthiness in our Abstraction Theorem (Section 5).

Open programs with ownership transfer. We start by introducing ingredients of the syntax of open programs. A ***predicate*** is a set of program states from State, and a ***parameterised predicate*** is a map from thread identifiers to predicates. We use the same symbols p, q, r for ordinary and parameterised predicates; it should always be clear from the context which one we mean. When p is a parameterised predicate, we write p_t for the predicate obtained by applying p to a thread t. We point out that both ordinary and parameterised predicates can be described syntactically, e.g., using separation logic assertions ([29] and Appendix C).

We describe possible ownership transfers between components with the aid of ***method specifications*** Γ, which are finite maps from method names to tuples (p, q, a) of parameterised predicates p, q and $a \in \{\mathsf{atomic}, \mathsf{nonatomic}\}$. Here p_t describes pieces of state transferred when thread t calls a method, and q_t those transferred at its return. The flag a describes whether the method is atomic or not. We sometimes ignore a in the tuples from Γ, writing them as Hoare triples: $\{p\}\ m\ \{q\} \in \Gamma$. We write $\Gamma \sqsubseteq \Gamma'$ when Γ and Γ' are identical except that Γ' classifies more methods as atomic.

As we explain below, to define the semantics of ownership transfers unambiguously, we require pre- and postconditions in method specifications to be precise. A predicate $r \in \mathcal{P}(\mathsf{State})$ is ***precise*** 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. Informally, a precise predicate carves out a unique piece of the heap. A parameterised predicate p is precise if for every t, the predicate p_t is precise.

An ***open program*** $\mathcal{P}$ is a program where the ground client or the implementations of some of the libraries may be missing:

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

The syntax of $\mathcal{P}$ differs from that of closed programs in two ways. First, open programs may include a hole $[-]$, which models a missing ground client. Second, we allow $\mathcal{P}$ to call methods that are not defined in it. Intuitively, such methods belong to libraries whose implementations are missing from $\mathcal{P}$. We often distinguish open programs with a ground client from those without one, and use the letter $\mathcal{C}$ for the former and $\mathcal{L}$ for the latter.

The environment an open program can interact with consists of two parts. First, the program can use some libraries with missing implementations. Second, the program itself might be an implementation of a library, providing some methods that can be called by an unspecified client. We refer to the former as its *imported libraries* and to the latter as its *external environment*. To specify possible interactions with these two types of environment, we use ***specified open programs*** or, simply, specified programs of the form $\Gamma \vdash \mathcal{P} : \Gamma'$. Here Γ includes the specifications of all the methods without implementations in $\mathcal{P}$. When $\mathcal{P}$ does not have a ground client, Γ' provides specifications for the methods in the open program that can be called by its external environment. Thus, it specifies the type of another open program $\mathcal{P}_0$ that can fill in the hole in $\mathcal{P}$ and behave as the client of the libraries that $\mathcal{P}$ defines.

Note that Γ' can include methods specified in Γ, in which case we require that they be given the same specifications in Γ' and Γ. For each method in Γ' implemented by $\mathcal{P}$, we require that its atomicity flag in Γ' have the value atomic precisely when the method is declared as such in $\mathcal{P}$. Finally, we also require that atomic methods implemented in $\mathcal{P}$ call only those methods in Γ that are there declared as atomic. We sometimes omit Γ and Γ' from a specified program when they are clear from the context.

When $\mathcal{P}$ implements all the libraries it uses, Γ can be $\emptyset$. When $\mathcal{P}$ contains a ground client, we require $\Gamma' = \emptyset$. Thus, closed programs of Section 2 can have specifications of the form $\emptyset \vdash S : \emptyset$.

Decomposing programs. We denote with $\mathcal{P}_1(\mathcal{P}_2)$ the result of filling the hole in an open program $\mathcal{P}_2$ with $\mathcal{P}_1$. We use this operation only when $\mathcal{P}_2$ contains a hole; $\mathcal{P}_1$ may or may not contain one. Thus, $\mathcal{P}_2$ defines the implementation of libraries, and $\mathcal{P}_1$ a client of these libraries, which could be another library implementation or a program with a ground client. When $\mathcal{P}_1$ and $\mathcal{P}_2$ have specifications $\Gamma' \vdash \mathcal{P}_1 : \Gamma''$ and $\Gamma \vdash \mathcal{P}_2 : \Gamma'$, the program $\mathcal{P}_1(\mathcal{P}_2)$ gets the specification $\Gamma \vdash \mathcal{P}_1(\mathcal{P}_2) : \Gamma''$. A common use of this decomposition is to represent a closed program $\emptyset \vdash S : \emptyset$ of Section 2 as $S = \mathcal{C}(\mathcal{L})$, where $\Gamma' \vdash \mathcal{C} : \emptyset$ and $\emptyset \vdash \mathcal{L} : \Gamma'$.

Traces. In the semantics we define below, a specified program $\Gamma \vdash \mathcal{P} : \Gamma'$ denotes a set of ***traces***, which are finite or infinite sequences of actions φ of the form:

$$\varphi ::= (t, c) \mid (t, \mathsf{call}\ m) \mid (t, \mathsf{ret}\ m) \mid (t, \mathsf{call}\ m(\theta)) \mid (t, \mathsf{ret}\ m(\theta))$$

where $t \in \mathsf{ThreadID}$, $c \in \mathsf{PComm}$, $\theta \in \mathsf{State}$ and $m \in \mathsf{Method}$. We denote the sets of corresponding types of actions with PAct, $\mathsf{CallAct}$, RetAct, $\mathsf{ECallAct}$ and $\mathsf{ERetAct}$, respectively, and their

union with Act. We write Trace for the set of all traces.

An action describes an atomic computation step made by a thread and can correspond to a primitive command, a method call or a return. Call and return actions from ECallAct and ERetAct represent interactions of the open program $\Gamma \vdash \mathcal{P} : \Gamma'$ with its environment: methods from Γ with unspecified implementation or unspecified clients calling methods in Γ'. These actions are annotated with the state that gets transferred between the program and the environment. In contrast, actions in CallAct and RetAct result from calls to and returns from methods internal to the open program.

Let $\mathsf{ACallAct} = \mathsf{CallAct} \cup \mathsf{ECallAct}$ and $\mathsf{ARetAct} = \mathsf{RetAct} \cup \mathsf{ERetAct}$. We put the subscript M on sets of actions CallAct, RetAct, ECallAct, ERetAct, ACallAct and ARetAct to restrict them to calls to or returns from methods in M. For instance, $\mathsf{CallAct}_M$ denotes the set of call actions involving methods from the set M. We let $\mathsf{CallRetAct}_M = \mathsf{CallAct}_M \cup \mathsf{RetAct}_M$ and define $\mathsf{ECallRetAct}_M$ and $\mathsf{ACallRetAct}_M$ similarly. We omit M when $M = \mathsf{Method}$.

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 a standard manner (Appendix A).

We represent a specified program $\Gamma \vdash \mathcal{P} : \Gamma'$ by a collection of CFGs. If $\mathcal{P}$ contains a ground 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 semantics below 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 ground 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 ground 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 an open program $\Gamma \vdash \mathcal{P} : \Gamma'$ represented by its CFG. Let Pos be the set of ***thread positions***—non-empty finite sequences of elements of Node. A thread position describes the call-stack of a thread: its last element describes the program point of the current command, and the rest give the return points for the methods called.

We define the set of program configurations as

$$\mathsf{Config} = \{\top\} \cup (\mathsf{ThreadID} \rightharpoonup_{fin} \mathsf{Pos}) \times \mathsf{State} \times \mathsf{Foot} \times (\mathcal{P}(\mathsf{ThreadID}))^+$$

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 third component tracks the footprint of the imported libraries that the open program $\mathcal{P}$ might communicate with; it is updated upon ownership transfers at calls to and returns from these libraries. Tracking the footprint allows us to define a more precise semantics, which is required for the proof of our Abstraction Theorem (see Section 5.2). The last component is a non-empty sequence of sets of thread identifiers, which describes the current *scheduling policy* and is used to implement the semantics of atomic.

Figure 2. Transition relation $\longrightarrow_{\Gamma,\mathcal{P},\Gamma'}$ for a specified program $\Gamma \vdash \mathcal{P} : \Gamma'$. The set M contains all the methods declared atomic in $\mathcal{P}$ and Γ. We omit the subscripts $\Gamma, \mathcal{P}, \Gamma'$ to avoid clutter.

$$\frac{t \in K \qquad \alpha, \theta, l \xrightarrow{\varphi}_t \alpha', \theta', l' \qquad \varphi \notin \mathsf{ACallRetAct}_M}{\mathsf{pc}[t : \alpha], \theta, l, \kappa K \xrightarrow{\varphi} \mathsf{pc}[t : \alpha'], \theta', l', \kappa K} \tag{4}$$

$$\frac{t \in K \qquad \alpha, \theta, l \xrightarrow{\varphi}_t \top}{\mathsf{pc}[t : \alpha], \theta, l, \kappa K \xrightarrow{\varphi} \top} \tag{5}$$

$$\frac{t \in K \qquad \alpha, \theta, l \xrightarrow{\varphi}_t \alpha', \theta', l' \qquad \varphi \in \mathsf{ACallAct}_M}{\mathsf{pc}[t : \alpha], \theta, l, \kappa K \xrightarrow{\varphi} \mathsf{pc}[t : \alpha'], \theta', l', \kappa K\{t\}} \tag{6}$$

$$\frac{t \in K \qquad \alpha, \theta, l \xrightarrow{\varphi}_t \alpha', \theta', l' \qquad \varphi \in \mathsf{ARetAct}_M}{\mathsf{pc}[t : \alpha], \theta, l, \kappa K \xrightarrow{\varphi} \mathsf{pc}[t : \alpha'], \theta', l', \kappa} \tag{7}$$

$$\frac{(v, c, v') \in T \qquad f^t_c(\theta) \neq \top \qquad \theta' \in f^t_c(\theta)}{\alpha v, \theta, l \xrightarrow{(t,c)}_t \alpha v', \theta', l} \tag{8}$$

$$\frac{(v, c, v') \in T \qquad f^t_c(\theta) = \top}{\alpha v, \theta, l \xrightarrow{(t,c)}_t \top} \tag{9}$$

$$\frac{(v, m, v') \in T \qquad m \notin \mathsf{dom}(\Gamma)}{\alpha v, \theta, l \xrightarrow{(t,\mathsf{call}\ m)}_t \alpha v' \mathsf{start}_m, \theta, l} \tag{10}$$

$$\frac{m \notin \mathsf{dom}(\Gamma) \qquad \alpha \neq v^t_{\mathsf{mgc}}}{\alpha \mathsf{end}_m, \theta, l \xrightarrow{(t,\mathsf{ret}\ m)}_t \alpha, \theta, l} \tag{11}$$

$$\frac{\begin{array}{c}(v, m, v') \in T \qquad \{p\}\, m\, \{q\} \in \Gamma \\ \theta = \theta' * \theta_p \qquad \theta_p \in p_t \qquad l * \delta(\theta_p) \neq \emptyset\end{array}}{\alpha v, \theta, l \xrightarrow{(t,\mathsf{call}\ m(\theta_p))}_t \alpha v v_m, \theta', l * \delta(\theta_p)} \tag{12}$$

$$\frac{(v, m, v') \in T \qquad \{p\}\, m\, \{q\} \in \Gamma \qquad \theta \notin \mathsf{State} * p_t}{\alpha v, \theta, l \xrightarrow{(t,\mathsf{call}\ m(\epsilon))}_t \top} \tag{13}$$

$$\frac{\{p\}\, m\, \{q\} \in \Gamma \qquad \theta_q \in q_t \qquad (\theta * \theta_q)\!\downarrow \qquad (l \setminus \delta(\theta_q))\!\downarrow}{\alpha v_m, \theta, l \xrightarrow{(t,\mathsf{ret}\ m(\theta_q))}_t \alpha, \theta * \theta_q, l \setminus \delta(\theta_q)} \tag{14}$$

$$\frac{\{p\}\, m\, \{q\} \in (\Gamma' - \Gamma) \qquad \theta_p \in p_t \qquad \delta(\theta) * l * \delta(\theta_p) \neq \emptyset}{v^t_{\mathsf{mgc}}, \theta, l \xrightarrow{(t,\mathsf{call}\ m(\theta_p))}_t v^t_{\mathsf{mgc}} \mathsf{start}_m, \theta * \theta_p, l} \tag{15}$$

$$\frac{\{p\}\, m\, \{q\} \in (\Gamma' - \Gamma) \qquad \theta = \theta' * \theta_q \qquad \theta_q \in q_t}{v^t_{\mathsf{mgc}} \mathsf{end}_m, \theta, l \xrightarrow{(t,\mathsf{ret}\ m(\theta_q))}_t v^t_{\mathsf{mgc}}, \theta', l} \tag{16}$$

$$\frac{\{p\}\, m\, \{q\} \in (\Gamma' - \Gamma) \qquad \theta \notin \mathsf{State} * q_t}{v^t_{\mathsf{mgc}} \mathsf{end}_m, \theta, l \xrightarrow{(t,\mathsf{ret}\ m(\epsilon))}_t \top} \tag{17}$$

$$\frac{\{p\}\, m\, \{q\} \in \Gamma' \cap \Gamma \qquad \theta_p \in p_t \qquad \delta(\theta) * l * \delta(\theta_p) \neq \emptyset}{v^t_{\mathsf{mgc}}, \theta, l \xrightarrow{(t,\mathsf{call}\ m(\theta_p))}_t v^t_{\mathsf{mgc}} v_m, \theta, l * \delta(\theta_p)} \tag{18}$$

$$\frac{\{p\}\, m\, \{q\} \in \Gamma' \cap \Gamma \qquad \theta_q \in q_t \qquad (l \setminus \delta(\theta_q))\!\downarrow}{v^t_{\mathsf{mgc}} v_m, \theta, l \xrightarrow{(t,\mathsf{ret}\ m(\theta_q))}_t v^t_{\mathsf{mgc}}, \theta, l \setminus \delta(\theta_q)} \tag{19}$$

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 2. The relation is defined by rules (4)–(7). Rules (4) and (5) pick a thread t from the rightmost set in the current scheduling policy, which describes the set of schedulable threads, and let it execute an atomic command. The execution of an atomic command by t is described using an auxiliary relation $\longrightarrow_{t,\Gamma,\mathcal{P},\Gamma'} : \mathsf{PConfig} \times \mathsf{Act} \times \mathsf{PConfig}$, where $\mathsf{PConfig} = (\mathsf{Pos} \times \mathsf{State} \times \mathsf{Foot}) \cup \{\top\}$ is the set

of program configurations projected to a single thread. According to rule (6), when a thread t calls an atomic method, a new set $\{t\}$ gets appended to the sequence defining the scheduling policy, thus making t the only schedulable thread. Rule (7) removes this set when the method returns. Thus, expressing the scheduling policy with a a sequence handles nested invocations of atomic methods.

The relation $\longrightarrow_{t,\Gamma,\mathcal{P},\Gamma'}$ is defined by rules (8)–(19). Rules (8)–(11) correspond to internal actions of $\mathcal{P}$: the execution of primitive commands, or calls to and returns from methods defined in $\mathcal{P}$. Note that, upon a method call, the return point is saved as a component in the new thread position, and the method starts executing from the corresponding starting node of its CFG. Upon a return, the return point is read from the current program counter.

Rules (12)–(19) concern interactions with the environment of $\mathcal{P}$. They package a semantics for programs with a ground client and library implementations without one into a single transition relation. We explain the semantics for these two cases separately.

Client-local semantics. Consider a specified open program $\Gamma \vdash \mathcal{C} : \emptyset$, where $\mathcal{C}$ contains a ground client. This open program represents a client program using some libraries with unspecified implementation. Our transition relation in Figure 2 provides a ***client-local*** semantics of $\mathcal{C}$ in the sense that it generates executions of this client assuming any behaviour of its imported libraries consistent with Γ. According to (12), when a thread t calls a method in Γ, it transfers the ownership of a piece of state satisfying the method precondition p_t to the library being called. Since p_t is precise (see above), this piece of state is determined uniquely. The rule also updates the footprint of the imported libraries accordingly. By (13), the semantics faults if the state to be transferred is not available. This ensures that the open program respects the specifications of the libraries it uses. We do not use the resulting faulty traces in the future, so we can annotate the transition with any actions; we chose $(\mathsf{ret}\ m(\epsilon))$ to be specific.

According to the CFG of $\mathcal{P}$, a method from Γ returns immediately after having been called. As stated in (14), instead of running the method implementation, the client receives the ownership of an arbitrary piece of state satisfying the postcondition q_t of the method instantiated with the current thread identifier t, which has to be compatible with the state of the client. As before, the footprint of the library is updated appropriately. Finally, rules (15)–(19) never become applicable for programs with a ground client.

Library-local semantics. Now consider a specified open program $\Gamma \vdash \mathcal{L} : \Gamma'$ where Γ' is not empty and $\mathcal{L}$ does not have a ground client. This program represents implementations of libraries, possibly using some other libraries with unspecified implementations. As before, when $\mathcal{L}$ calls methods from the imported libraries, these behave in an arbitrary way consistent with Γ, as per rules (12)–(14). Rules (15)–(19), which are applicable to programs without ground clients, give a ***library-local*** semantics to the program $\mathcal{L}$. The semantics can be thought of as running the library under its *most general client*, which reproduces all possible library behaviours under any clients. This property is formalised by Lemma 12 (Decomposition) in Section 5.2. Assuming that every thread t in the program starts at the position v^t_{mgc}, it executes an infinite loop, repeatedly invoking arbitrary methods implemented in the library $\mathcal{L}$ or unimplemented, but specified in Γ. According to rule (15), when a method implemented by $\mathcal{L}$ gets called, the library receives the ownership of any state consistent with the method precondition. According to (16), after the method returns, the library has to give up the piece of state satisfying its postcondition. As before, this piece is defined uniquely, because postconditions are precise. When such a piece of state is not available, the program faults, as described by rule (17). This ensures that the library respects the contract with its client. Finally, the remaining rules (18) and (19) describe a call to and a return from a method that is not implemented in $\mathcal{L}$, but specified by Γ and Γ'. The computation proceeds as in the case of methods defined in $\mathcal{L}$, except that the ownership transfer happens between the external environment and the imported libraries, rather than the environment and $\mathcal{L}$.

Note that rules (15) and (18) have a side condition ensuring that the footprint of the state transferred from the environment is compatible with both the state of $\mathcal{L}$ and the state of its imported libraries. This means that the library-local semantics generates the behaviours of $\mathcal{L}$ only in the *environment that respects the notion of ownership*, i.e., does not attempt to transfer pieces of state it cannot possibly own at calls to $\mathcal{L}$. A theorem in Section 5.2 (Lemma 12) implies that every instantiation of the environment with a ground client satisfies this constraint, and thus, it does not result in a loss of generality. Including the compatibility with the state of imported libraries into the constraint is needed for the proof of our Abstraction Theorem and is the reason for tracking the footprint of such libraries in the semantics.

Trace interpretation. Our operational semantics induces the trace interpretation of specified open programs $\Gamma \vdash \mathcal{P} : \Gamma'$. For a finite trace τ and $\varsigma, \varsigma' \in \mathsf{Config}$ we write $\varsigma \xrightarrow{\tau}{}^*_{\Gamma,\mathcal{P},\Gamma'} \varsigma'$ if there exists a corresponding derivation of τ using $\longrightarrow_{\Gamma,\mathcal{P},\Gamma'}$. Similarly, for an infinite trace τ and $\varsigma \in \mathsf{Config}$ we write $\varsigma \xrightarrow{\tau}{}^\omega_{\Gamma,\mathcal{P},\Gamma'} -$ to mean the existence of an infinite τ-labelled computation from ς according to our semantics. We denote with 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 ground 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:

$$\begin{aligned}[\![\Gamma \vdash \mathcal{P} : \Gamma']\!] = \{(\theta_0, l_0, \tau) \mid\ & \theta_0 \in \mathsf{State} \wedge l_0 \in \mathsf{Foot} \wedge \\ & \mathsf{pc}_0 \in \mathsf{PC}_0 \wedge ((\mathsf{pc}_0, \theta_0, l_0, \mathsf{ThreadID}) \xrightarrow{\tau}{}^\omega_{\Gamma,\mathcal{P},\Gamma'} - \vee \\ & \exists \varsigma \in \mathsf{Config} - \{\top\}.\, (\mathsf{pc}_0, \theta_0, l_0, \mathsf{ThreadID}) \xrightarrow{\tau}{}^*_{\Gamma,\mathcal{P},\Gamma'} \varsigma)\}.\end{aligned}$$

An element in $[\![\mathcal{P}]\!]$ records an initial state θ_0, an initial footprint l_0 of imported libraries, and a finite or infinite execution trace, which does not have to be maximal. For a specified open program $\Gamma \vdash \mathcal{P} : \Gamma'$, we often consider a set $\mathcal{I} \subseteq \{(\theta, l) \in \mathsf{State} \times \mathsf{Foot} \mid \delta(\theta) * l \neq \emptyset\}$ defining its initial configurations. We call $\mathcal{I}$ an ***initial condition*** of $\mathcal{P}$ and let

$$[\![(\Gamma \vdash \mathcal{P} : \Gamma'), \mathcal{I}]\!] = \{(\theta_0, l_0, \tau) \in [\![\Gamma \vdash \mathcal{P} : \Gamma']\!] \mid (\theta_0, l_0) \in \mathcal{I}\}.$$

Note that we require the states of the program and its imported libraries in initial conditions to be compatible. In the following we only use initial conditions $\mathcal{I}$ where the footprints of imported libraries are downwards-closed with respect to $\preceq$:

$$\forall (\theta, l) \in \mathcal{I}.\, \forall l'.\, l' \preceq l \Rightarrow (\theta, l') \in \mathcal{I}. \tag{20}$$

This assumption simplifies the formulation of our results.

We use the standard notation for traces: ε is the empty trace, $\tau(i)$ is the i-th action in the trace τ, $|\tau|$ is the length of the trace τ ($|\tau| = \omega$ if τ is infinite), and $\tau|_t$ is the projection of τ to actions of thread t. We call a trace τ ***well-formed*** if calls and returns in $\tau|_t$ are well-nested for every $t \in \mathsf{ThreadID}$. It is easy to check that traces generated by the semantics of open programs are well-formed.

Safety of open programs. A program $\Gamma \vdash \mathcal{P} : \Gamma'$ is ***safe*** at (θ_0, l_0) if it is not the case that $(\mathsf{pc}_0, \theta_0, l_0, \mathsf{ThreadID}) \xrightarrow{\tau}{}^*_{\Gamma,\mathcal{P},\Gamma'} \top$ for some τ. The program $\mathcal{P}$ is ***safe for an initial condition*** $\mathcal{I}$, if it is safe at (θ_0, l_0) for all $(\theta_0, l_0) \in \mathcal{I}$.

Because of the locality property (2), 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. According to rules (13) and (17), calls to methods in Γ and returns from methods in $\Gamma' - \Gamma$ fault when the piece of state they have to transfer to the environment is not available. Thus, the safety of the program also ensures that it respects the contract with its environment.

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, the safety of an open program representing a client of a library formalises the notion of *client healthiness* required in the

Abstraction Theorem (Section 5). We also define the linearizability relation only between safe libraries (Section 4). In practice, the safety of a program is established using a program logic, one of which we present in Appendix C.

We now illustrate how the client- and library-local semantics introduced here allow us to reason modularly about components of a program. Let us define a partial $\otimes$ operation on elements (θ_1, l_1) and (θ_2, l_2) of initial conditions as follows: $(\theta_1, l_1) \otimes (\theta_2, l_2) = (\theta_1 * \theta_2, l_2)$, if $(\theta_1 * \theta_2)\downarrow$ and $l_1 = \delta(\theta_2) * l_2$; undefined otherwise. We lift $\otimes$ to initial conditions pointwise. When $\mathcal{I}_L$ and $\mathcal{I}_P$ are respective initial conditions of a library $\mathcal{L}$ and its client $\mathcal{P}$, $\mathcal{I}_P \otimes \mathcal{I}_L$ computes the initial condition of their combination $\mathcal{P}(\mathcal{L})$. An initial state is obtained by combining those of $\mathcal{P}$ and $\mathcal{L}$, with the initial state of $\mathcal{P}$ picked from a pair with the footprint matching the initial condition of $\mathcal{L}$. Since the imported libraries of $\mathcal{P}(\mathcal{L})$ are identical to those of $\mathcal{L}$, the initial footprints of $\mathcal{P}(\mathcal{L})$ and $\mathcal{L}$ are the same.

Theorem 3 (Information hiding). *Consider* $\Gamma_0 \vdash \mathcal{L} : \Gamma_1$ *and* $\Gamma_1 \vdash \mathcal{P} : \Gamma_2$. *If* $\mathcal{L}$ *and* $\mathcal{P}$ *are safe for their respective initial conditions* $\mathcal{I}_L$ *and* $\mathcal{I}_P$, *then so is* $\mathcal{P}(\mathcal{L})$ *for* $\mathcal{I}_P \otimes \mathcal{I}_L$.

We defer the proof to Appendix B and note that an obvious generalisation of the theorem to partial correctness properties holds as well. The theorem allows hiding the state of the library while reasoning about the client and vice versa. In this sense it generalises sequential proof rules for information hiding [25]; see Section 7.

Summary. So far, we have presented a novel semantics for specified open programs interacting with their environment via ownership transfers. We have also demonstrated, via Theorem 3 that this semantics can be used to decompose the reasoning about a given program $\mathcal{P}(\mathcal{L})$ into reasoning about its constituent components $\mathcal{P}$ and $\mathcal{L}$. We note that the client-local semantics of $\mathcal{P}$ and library-local one of $\mathcal{L}$ can be viewed as defining a compositional semantics of the program $\mathcal{P}(\mathcal{L})$. Of course, to use this semantics we need to connect it to a standard semantics of closed programs. Let us first note that for such programs, the semantics in Figure 2 with the initial library footprint $\delta(\epsilon)$ is just a standard operational semantics of the language of Section 2. In Section 5.2, we present a theorem showing that the compositional semantics of a closed program $\mathcal{P}(\mathcal{L})$, given by the client- and library-local semantics of its components $\mathcal{P}$ and $\mathcal{L}$, is sound (Lemma 12) with respect to the standard one. In particular, Theorem 3 is a corollary of this result. Finally, we can reason about the semantics of open programs using standard logics (Appendix C).

As we argue in Section 6, just decomposing the verification of a program as it is given is not enough: in many cases, we first need to replace library implementations in it with simpler ones. In the following sections we develop a technique allowing this.

4. Linearizability in the presence of ownership transfer

Specifications of concurrent libraries are usually given by their *abstract implementations*, most often consisting of atomically implemented methods with the state of the library represented by an abstract data type. The classical notion of linearizability [16] fixes a correspondence between an implementation of a concurrent library and such a specification. We now generalise it to our setting.

We define linearizability between specified open programs $\Gamma \vdash \mathcal{L} : \Gamma'$ without a ground client, together with their initial conditions $\mathcal{I}$. A program $\mathcal{L}$ represents an implementation of libraries, which provide methods in Γ' to their client libraries or programs, and which can also invoke methods that are not implemented, but specified in Γ. When defining linearizability, we are not interested in internal steps recorded in library traces from $[\![(\Gamma \vdash \mathcal{L} : \Gamma'), \mathcal{I}]\!]$, but only in the interactions of the libraries with their environment. We record such interactions using ***histories***, which are traces consisting only of call and return actions with ownership transfer, i.e., those in $\mathsf{ECallRetAct}$. Let $\mathsf{History}$ be the set of all histories.

In a well-formed trace τ, we can always identify pairs of calls and returns that correspond to outermost invocations of methods from a given set M in the trace $\tau|_t$ of some thread t. For example, when $M = \{m_1, m_2\}$, such calls and returns in the trace

$$(1, \mathsf{call}\ m_1)\,(2, \mathsf{call}\ m_3)\,(1, \mathsf{call}\ m_2(\theta_1))\,(2, \mathsf{call}\ m_2(\theta_2))\\ (2, \mathsf{ret}\ m_2(\theta_3))(1, \mathsf{ret}\ m_2(\theta_4))\,(1, \mathsf{ret}\ m_1)\,(2, \mathsf{ret}\ m_3)$$

are $(1, \mathsf{call}\ m_1), (1, \mathsf{ret}\ m_1), (2, \mathsf{call}\ m_2(\theta_2)), (2, \mathsf{ret}\ m_2(\theta_3))$. We define $\mathsf{history}_M(\tau)$ as the projection of τ to outermost calls and returns in $\mathsf{ECallRetAct}_M$. We lift it to sets of traces pointwise, so that the set of histories of a library $\Gamma \vdash \mathcal{L} : \Gamma'$ with an initial condition $\mathcal{I}$ is given as follows: for $M = \mathsf{dom}(\Gamma')$,

$$\mathsf{history}_M(\mathcal{L}, \mathcal{I}) = \{(\theta_0, l_0, \mathsf{history}_M(\tau)) \mid (\theta_0, l_0, \tau) \in [\![\mathcal{L}, \mathcal{I}]\!]\}.$$

Definition 4. *The* ***linearizability relation*** *is a binary relation* $\sqsubseteq$ *on histories defined as follows:* $H \sqsubseteq H'$ *if (i)* $\forall t \in \mathsf{ThreadID}.\ H|_t = H'|_t$ *and (ii) there is a bijection* $\pi\colon \{1, \ldots, |H|\} \to \{1, \ldots, |H'|\}$ *such that* $\forall i.\ H(i) = H'(\pi(i))$ *and*

$$\forall i, j.\ i < j \wedge H(i) \in \mathsf{ERetAct} \wedge H(j) \in \mathsf{ECallAct} \Rightarrow \pi(i) < \pi(j).$$

That is, the history H' linearizes the history H when it is a permutation of the latter preserving the order of actions within threads and non-overlapping method invocations.

The original definition of linearizability [16] only defines it on sets of histories as above, without taking into account library implementations that generate them. Performing library abstraction, however, requires us to consider the implementations being replaced. The semantics of open programs of Section 3 provides a way to generate the set of histories of libraries interacting with their environment via ownership transfers, which allows us to lift the notion of linearizability to library implementations as follows.

Definition 5. *Consider* $\Gamma \vdash \mathcal{L}_1 : \Gamma_1'$ *and* $\Gamma \vdash \mathcal{L}_2 : \Gamma_2'$ ***safe for*** $\mathcal{I}_1$ *and* $\mathcal{I}_2$, *respectively, and assume* $\Gamma_1' \sqsubseteq \Gamma_2'$. *We say that* $(\mathcal{L}_2, \mathcal{I}_2)$ ***linearizes*** $(\mathcal{L}_1, \mathcal{I}_1)$, *written* $(\mathcal{L}_1, \mathcal{I}_1) \sqsubseteq (\mathcal{L}_2, \mathcal{I}_2)$, *if*

$$\forall (\theta_1, l_1, H_1) \in \mathsf{history}_{\mathsf{dom}(\Gamma_1')}(\mathcal{L}_1, \mathcal{I}_1).\ \exists (\theta_2, l_2, H_2) \in \\ \mathsf{history}_{\mathsf{dom}(\Gamma_2')}(\mathcal{L}_2, \mathcal{I}_2).\ H_1 \sqsubseteq H_2 \wedge \theta_1 \succeq \theta_2 \wedge l_1 \succeq l_2.$$

Thus, $(\mathcal{L}_2, \mathcal{I}_2)$ linearizes $(\mathcal{L}_1, \mathcal{I}_1)$ if every behaviour of the latter in the library-local semantics may be reproduced in a linearized form by the former without requiring more memory resources. The requirement $\Gamma_1' \sqsubseteq \Gamma_2'$ in Definition 5 formalises the intuition that the linearization of a library has a coarser granularity of atomic actions than the original one. We now explain the motivation behind Definitions 4 and 5 in detail and highlight their novel features.

Ownership transfer. Definition 4 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 [16]: 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.

However, this treatment of ownership transfer is more subtle than might seem at first sight. The fact that our definition can be obtained straightforwardly from the classical one is made possible by the carefully crafted library-local semantics from which we generate histories. To see why this is the case, observe that some histories may never be generated by the library-local semantics. For example, consider the set of states RAM and the history

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

The history describes *all* the interactions between a library and its 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. If the history were generated by the semantics of the library, the latter transition would clearly be impossible, as the cell would be owned by the library, not by the client, before the second transfer. This is ensured by the fact that the library-local

semantics only considers environments that respect the notion of ownership: rule (15) in the semantics of Section 3 never transfers cells incompatible with the current library state.

Our Abstraction Theorem relies crucially on the fact that the histories considered are generated from the library-local semantics and thus account for ownership transfer properly (see Section 5 for discussion). The notion of such histories is formalised as follows. Let us define a function $\mathsf{run} : \mathsf{History} \times \mathsf{Foot} \rightharpoonup \mathsf{Foot}$, which tracks how the library footprint changes during a computation with a given history. It is defined for finite histories inductively:

$$\begin{array}{ll}
\mathsf{run}(\varepsilon, l) & = l; \\
\mathsf{run}(\varphi\tau, l) & = \mathsf{run}(\tau, l * \delta(\theta)), \text{ if } \varphi = (_, \mathsf{call}\ _(\theta)) \wedge l * \delta(\theta) \neq \emptyset; \\
\mathsf{run}(\varphi\tau, l) & = \mathsf{run}(\tau, l \setminus \delta(\theta)), \text{ if } \varphi = (_, \mathsf{ret}\ _(\theta)) \wedge \delta(\theta) \preceq l; \\
\mathsf{run}(\varphi\tau, l) & = \text{undefined}, \quad \text{otherwise.}
\end{array}$$

Definition 6. *The history H is **well-balanced** from l, when it is well-formed and $\mathsf{run}(H_0, l)$ is defined (in particular, distinct from $\emptyset$) for all finite prefixes H_0 of H.*

As the following proposition (Appendix B) shows, histories generated from the library-local semantics are well-balanced.

Proposition 7. *Assume $\Gamma \vdash \mathcal{L} : \Gamma'$ safe for $\mathcal{I}$. For all $(\theta, l, H) \in \mathsf{history}_{\mathsf{dom}(\Gamma')}(\mathcal{L}, \mathcal{I})$, the history H is well-balanced from $\delta(\theta) * l$.*

Multiple libraries. Definition 4 requires us to preserve the order of non-overlapping operations even between methods of different libraries. In the case when libraries do not interact and every one of them is linearizable, this condition comes for free: one can always find a linearization of a history preserving the order of all non-overlapping operations given linearizations of its projections to every single library [16]. In the case when libraries may interact, this requirement has to be built into the notion of linearizability. The resulting definition is the same as would result if we merged all the libraries involved into a single one. However, as we show in Section 5.1, by carefully structuring the process of abstracting several interacting libraries using a strong enough Abstraction Theorem, we can avoid having to merge several concrete library implementations. Instead, we can consider only one concrete implementation together with several abstract ones.

Memory requirements. Definition 5 requires the initial state of the abstract library implementation to be smaller than the initial state of the concrete one. The requirement is needed to ensure that all behaviours of a client using the concrete implementation be reproducible when it uses the abstract one instead. Namely, if the client is itself an open program interacting with an external environment, the memory whose ownership can be transferred to the client from the environment has to be compatible with the footprint of the library. If the footprint of the abstract implementation is smaller than that of the concrete one, any ownership transfers from the environment external to the client using the latter can be reproduced if the client uses the former instead. Since histories of ownership transfers for the two library implementations are related by linearizability, it is sufficient to require the condition only of initial states.

This requirement on memory usage has been previously used in data refinement [13]. It can be relaxed to allow the abstract library implementation to use more memory cells than the concrete one, provided that these additional 'abstract' cells cannot be transferred to and from an external environment. This is guaranteed if they cannot occur in method specifications. Our results can be easily extended to such a setting, which we do not formalise so as not to obscure the presentation. With this extension, the restriction on memory usage does not pose problems in practice, as demonstrated by our examples (Section 6).

Infinite histories. While the classical definition of linearizability considers only finite histories, we also consider infinite ones, following a generalisation by Gotsman and Yang [14]. This allows library specifications to express liveness properties and avoids pending call completions in the definition of linearizability [16].

Establishing the new linearizability. We have developed a logic for proving the proposed notion of linearizability, which generalises an existing proof system [32] based on separation logic [29] to the setting with ownership transfer. The logic uses the usual method of proving linearizability based on linearization points [1, 16, 32] 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.

5. Abstraction Theorem

Given the notions of client and library safety from Section 3 and linearizability from Section 4, we can now state and prove the central technical result of this paper—the Abstraction Theorem.

For a well-formed trace $\tau \in \mathsf{Trace}$ and a set of methods M, we define the following projections of τ:

- $\mathsf{client}_M(\tau)$ to actions other than those inside an outermost invocation of a method in M, including the outermost calls to and returns from M (see Section 4);
- $\mathsf{visible}_M(\tau)$ to the same actions, but excluding the outermost calls to and returns from methods in M;
- $\mathsf{lib}_M(\tau)$ to actions other than those outside an outermost invocation of a method in M, including the outermost calls and returns.

We lift these operations to sets of traces pointwise.

The Abstraction Theorem shows that replacing a library used by a client with its linearization leaves all the original client behaviours reproducible modulo the following notion of trace refinement. Note that in the following this notion is used on traces without internal library actions, obtained as results of $\mathsf{client}_M(\cdot)$, where M is the set of methods implemented by the library.

Definition 8. *A trace $\tau \in \mathsf{Trace}$ **refines** another trace τ' with respect to a set of methods M, written $\tau \lhd_M \tau'$, if (i) $\tau|_t = \tau'|_t$ for every $t \in \mathsf{ThreadID}$ and (ii) there exists a bijection $\pi : \{1, \ldots, |\tau|\} \to \{1, \ldots, |\tau'|\}$ such that $\forall i.\, \tau(i) = \tau'(\pi(i))$ and*

$$\begin{array}{l}
\forall i, j.\, (i < j \wedge ((\tau(i), \tau(j) \in \mathsf{Act} - \mathsf{ACallRetAct}_M) \vee \\
\quad (\tau(i) \in \mathsf{Act} - \mathsf{ACallRetAct}_M \wedge \tau(j) \in \mathsf{ACallAct}_M) \vee \\
(\tau(i) \in \mathsf{ARetAct}_M \wedge \tau(j) \in \mathsf{Act} - \mathsf{ARetAct}_M))) \Rightarrow \pi(i) < \pi(j).
\end{array}$$

According to the definition, τ' is obtained from τ by a permutation π that preserves the order of certain pairs of actions. In particular, π preserves the order of actions within threads and actions of the client (except calls to and returns from M). Hence, $\mathsf{client}_M(\tau) \lhd_M \mathsf{client}_M(\tau')$ implies $\mathsf{visible}_M(\tau) = \mathsf{visible}_M(\tau')$, so $\lhd_M$ preserves any linear-time temporal property over trace projections to client actions. Additionally, π preserves the order of a client action followed by a call to a method in M and a return from a method in M followed by a client action. As we show in Section 5.1, this property is needed for abstracting multiple libraries. Finally, it preserves the order of a return action followed by a call action, like in the definition of linearizability (Definition 4). The following theorem, proved in Section 5.2 and Appendix B, states our abstraction result.

Theorem 9 (Abstraction). *Consider $\Gamma_0 \vdash \mathcal{L}_1 : \Gamma_1$, $\Gamma_0 \vdash \mathcal{L}_2 : \Gamma_1'$, $\Gamma_1 \vdash \mathcal{P} : \Gamma_2$ and $\Gamma_1' \vdash \mathcal{P} : \Gamma_2'$ safe for $\mathcal{I}_1$, $\mathcal{I}_2$, $\mathcal{I}$ and $\mathcal{I}$, respectively. Let $\Gamma_1 \sqsubseteq \Gamma_1'$, $\Gamma_2 \sqsubseteq \Gamma_2'$. If $(\mathcal{L}_1, \mathcal{I}_1) \sqsubseteq (\mathcal{L}_2, \mathcal{I}_2)$, then*

$$\begin{array}{l}
\forall (\theta_1, l_1, \tau_1) \in [\![\mathcal{P}(\mathcal{L}_1), \mathcal{I} \otimes \mathcal{I}_1]\!].\, \exists (\theta_2, l_2, \tau_2) \in [\![\mathcal{P}(\mathcal{L}_2), \mathcal{I} \otimes \mathcal{I}_2]\!]. \\
\mathsf{client}_{\mathsf{dom}(\Gamma_1)}(\tau_1) \lhd_{\mathsf{dom}(\Gamma_1)} \mathsf{client}_{\mathsf{dom}(\Gamma_1')}(\tau_2) \wedge \theta_1 \succeq \theta_2 \wedge l_1 \succeq l_2.
\end{array}$$

From the properties of trace refinement noted above, we get

Corollary 10. *Under the conditions of Theorem 9, we have*

$$\begin{array}{l}
\mathsf{visible}_{\mathsf{dom}(\Gamma_1)}(\{\tau_1 \mid (\theta_1, l_1, \tau_1) \in [\![\mathcal{P}(\mathcal{L}_1), \mathcal{I} \otimes \mathcal{I}_1]\!]\}) \subseteq \\
\qquad \mathsf{visible}_{\mathsf{dom}(\Gamma_1')}(\{\tau_2 \mid (\theta_2, l_2, \tau_2) \in [\![\mathcal{P}(\mathcal{L}_2), \mathcal{I} \otimes \mathcal{I}_2]\!]\}).
\end{array}$$

According to Corollary 10, while reasoning about a client $\mathcal{P}(\mathcal{L}_1)$ of a library $\mathcal{L}_1$, we can soundly replace $\mathcal{L}_1$ with a sim-

pler library $\mathcal{L}_2$ linearizing $\mathcal{L}_1$: if a linear-time property over client actions holds of $\mathcal{P}(\mathcal{L}_2)$, it will also hold of $\mathcal{P}(\mathcal{L}_1)$. In practice, we are usually interested in ***atomicity abstraction*** (see, e.g., [20]), a special case of this transformation when methods in $\mathcal{L}_2$ are atomic.

The requirement that $\mathcal{P}$ be safe in Theorem 9 restricts its applicability to healthy clients that do not access library internals. This requirement, as well as that of linearizability, can be established using program logics as described in Appendices C and D. However, we emphasise that the formulation of the theorem is not tied to a particular logic: the semantics of Section 3 provides an interface prescribing what a logic being used needs to establish.

5.1 Abstracting multiple libraries

Note that the programs $\mathcal{P}(\mathcal{L}_1)$ and $\mathcal{P}(\mathcal{L}_2)$ in our Abstraction Theorem do not have to be closed. In particular, $\mathcal{P}$ might itself be a library implementation that uses methods provided by $\mathcal{L}_1$ or $\mathcal{L}_2$. The following corollary of Theorem 9 can then be used to simplify the proof of the linearizability of $\mathcal{P}$.

Corollary 11. *Consider* $\Gamma_0 \vdash \mathcal{L}_1 : \Gamma_1$, $\Gamma_0 \vdash \mathcal{L}'_1 : \Gamma'_1$, $\Gamma_1 \vdash \mathcal{L}_2 : \Gamma_2$, *and* $\Gamma'_1 \vdash \mathcal{L}_2 : \Gamma'_2$ *safe for* $\mathcal{I}_1$, $\mathcal{I}'_1$, $\mathcal{I}_2$ *and* $\mathcal{I}_2$, *respectively. Assume* $\Gamma_1 \sqsubseteq \Gamma'_1$ *and* $\Gamma_2 \sqsubseteq \Gamma'_2$. *Then*

$(\mathcal{L}_1, \mathcal{I}_1) \sqsubseteq (\mathcal{L}'_1, \mathcal{I}'_1) \Rightarrow (\mathcal{L}_2(\mathcal{L}_1), \mathcal{I}_2 \otimes \mathcal{I}_1) \sqsubseteq (\mathcal{L}_2(\mathcal{L}'_1), \mathcal{I}_2 \otimes \mathcal{I}'_1)$.

Proof. Consider $(\theta_1, l_1, \tau_1) \in [\![\mathcal{L}_2(\mathcal{L}_1), \mathcal{I}_2 \otimes \mathcal{I}_1]\!]$. By Theorem 9, for some $(\theta_2, l_2, \tau_2) \in [\![\mathcal{L}_2(\mathcal{L}'_1), \mathcal{I}_2 \otimes \mathcal{I}'_1]\!]$ we have

$\mathsf{client}_{\mathsf{dom}(\Gamma_1)}(\tau_1) \lhd_{\mathsf{dom}(\Gamma_1)} \mathsf{client}_{\mathsf{dom}(\Gamma'_1)}(\tau_2) \wedge \theta_1 \succeq \theta_2 \wedge l_1 \succeq l_2$.

Then by Definition 8, $\mathsf{history}_{\mathsf{dom}(\Gamma_2)}(\tau_1) \sqsubseteq \mathsf{history}_{\mathsf{dom}(\Gamma'_2)}(\tau_2)$. □

The step of the proof going from $\lhd_{\mathsf{dom}(\Gamma_1)}$ to $\sqsubseteq$ is subtle: it relies crucially on the second and third clauses specifying order preservation in Definition 8, which are not needed for proving Corollary 10. Namely, we need to ensure that $\lhd_{\mathsf{dom}(\Gamma_1)}$ preserves the order between non-overlapping invocations of methods implemented by $\mathcal{L}_1$ and $\mathcal{L}_2$. Since the latter are client actions from the the point of view of $\mathcal{L}_1$, this is justified by the additional clauses in the definition of $\lhd$, which guarantee the preservation of the order between a client action followed by a call to a method implemented in $\mathcal{L}_1$, and a return from such a method followed by a client action. Hence, abstracting multiple libraries requires a stronger statement of the Abstraction Theorem than abstracting a single one.

The corollary allows us to prove linearizability compositionally in the structure of a library. If we prove that a library $\mathcal{L}_1$ is linearized by an atomically implemented library $\mathcal{L}'_1$, then proving the linearizability of $\mathcal{L}_2(\mathcal{L}_1)$ can be simplified. Since $\sqsubseteq$ is transitive, instead of proving the linearizability of $\mathcal{L}_2(\mathcal{L}_1)$ by $\mathcal{L}'_2(\mathcal{L}'_1)$ for some $\mathcal{L}'_2$ directly, we can instead prove that the latter linearizes a simpler library $\mathcal{L}_2(\mathcal{L}'_1)$. This can be generalised to abstracting multiple libraries. Consider a program

$$\mathsf{let}\ L_1\ \mathsf{in}\ \mathsf{let}\ L_2\ \mathsf{in}\ \mathsf{let}\ L_3\ \mathsf{in}\ \ldots\ \mathsf{let}\ L_k\ \mathsf{in}\ [-]$$

Let $\mathcal{L}_i$ be $(\mathsf{let}\ L_i\ \mathsf{in}\ [-])$. We first find a linearization $\mathcal{L}'_1 = (\mathsf{let}\ L'_1\ \mathsf{in}\ [-])$ of $\mathcal{L}_1$, usually with L'_1 implemented atomically. By Corollary 11, $\mathcal{L}_2(\mathcal{L}_1)$ is linearized by $\mathcal{L}_2(\mathcal{L}'_1)$. We then prove the linearizability of the latter by $\mathcal{L}'_2(\mathcal{L}'_1)$ for some $\mathcal{L}'_2 = (\mathsf{let}\ L'_2\ \mathsf{in}\ [-])$. Then by Corollary 11 and the transitivity of $\sqsubseteq$, $\mathcal{L}_3(\mathcal{L}_2(\mathcal{L}_1))$ is linearized by $\mathcal{L}_3(\mathcal{L}'_2(\mathcal{L}'_1))$. Continuing with this process, we can abstract all the libraries in the program, showing it is linearized by

$$\mathsf{let}\ L'_1\ \mathsf{in}\ \mathsf{let}\ L'_2\ \mathsf{in}\ \mathsf{let}\ L'_3\ \mathsf{in}\ \ldots\ \mathsf{let}\ L'_k\ \mathsf{in}\ [-]$$

Note that, even though the above process requires considering several libraries together (e.g., $\mathcal{L}'_1$, $\mathcal{L}'_2$, $\mathcal{L}_3$), we only have to consider an implementation of one them ($\mathcal{L}_3$), together with specifications of the libraries it calls ($\mathcal{L}'_1$, $\mathcal{L}'_2$). We never have to consider concrete implementations of several libraries together, which would make reasoning non-modular.

5.2 Proof outline

The proof of Theorem 9 is both complicated and subtle. For this reason, here we provide a proof outline, explaining the method we followed, stating core lemmas and highlighting non-trivial places in the proof. To prove Theorem 9, we need to transform a trace τ_1 of $\mathcal{P}(\mathcal{L}_1)$ into a trace τ_2 of $\mathcal{P}(\mathcal{L}_2)$ with client projections related by $\lhd_{\mathsf{dom}(\Gamma_1)}$. Whereas client actions only get permuted when going from τ_1 to τ_2, library actions get changed completely. To replace library actions, we use the semantics in Section 3, which provides the trace interpretation of $\mathcal{L}_1$, $\mathcal{L}_2$, $\mathcal{P}$ and their compositions.

To transform τ_1 into τ_2, we first show that a trace of $\mathcal{P}(\mathcal{L}_1)$, such as τ_1, generates two traces agreeing on the history of methods in $\mathsf{dom}(\Gamma_1)$: ξ_1 in the semantics of $\Gamma_0 \vdash \mathcal{L}_1 : \Gamma_1$ and η_1 in the semantics of $\Gamma_1 \vdash \mathcal{P} : \Gamma_2$ (Lemma 12). Note that the trace η_1 of $\mathcal{P}$ thus constructed excludes the internal library actions. Since $(\mathcal{L}_1, \mathcal{I}_1) \sqsubseteq (\mathcal{L}_2, \mathcal{I}_2)$, there exists a history H_2 of $\mathcal{L}_2$ with respect to methods in $\mathsf{dom}(\Gamma'_1)$ linearizing $\mathsf{history}_{\mathsf{dom}(\Gamma_1)}(\eta_1)$. We then show that η_1 can be transformed into a trace η_2 of $\mathcal{P}$ satisfying $\eta_1 \lhd_{\mathsf{dom}(\Gamma_1)} \eta_2$ and also having the target history H_2 of $\mathcal{L}_2$ (Lemma 13). Finally, we show that the library-local trace ξ_2 generating this history of $\mathcal{L}_2$ can be composed with η_2 to yield the desired trace τ_2 of $\mathcal{P}(\mathcal{L}_2)$ (Lemma 14). This proof scheme can be described mnemonically as 'decompose, rearrange, compose'.

We now formulate the necessary lemmas, proved in Appendix B. For a set M of methods, let erase_M be a function on well-formed traces that erases the state annotations θ of all calls to and returns from methods in M. Let ierase_M be its variant that erases the annotations in all calls to and returns from M, except those that correspond to outermost invocations of *any* methods. The following operation combines traces of $\mathcal{L}$ and $\mathcal{P}$ into those of $\mathcal{P}(\mathcal{L})$ by interleaving their internal library and client actions.

$$\begin{aligned}(\theta_1, l_1, \xi) \circ_{\Gamma_0,\Gamma_1,\Gamma_2} (\theta_2, l_2, \eta) = \{(\theta, l, \tau) \mid (\theta, l) = (\theta_2, l_2) \otimes (\theta_1, l_1)\\ \wedge\ \mathsf{history}_{\mathsf{dom}(\Gamma_1)}(\xi) = \mathsf{history}_{\mathsf{dom}(\Gamma_1)}(\eta)\\ \wedge\ \mathsf{erase}_{\mathsf{dom}(\Gamma_1)-\mathsf{dom}(\Gamma_0)}(\mathsf{lib}_{\mathsf{dom}(\Gamma_1)}(\tau)) = \mathsf{erase}_{\mathsf{dom}(\Gamma_1)-\mathsf{dom}(\Gamma_0)}(\xi)\\ \wedge\ \mathsf{client}_{\mathsf{dom}(\Gamma_1)}(\tau) = \mathsf{ierase}_{(\mathsf{dom}(\Gamma_1)-\mathsf{dom}(\Gamma_0))\cap\mathsf{dom}(\Gamma_2)}(\eta)\}.\end{aligned}$$

It requires that the traces being combined agree on the history of the interface methods from Γ_1 and have compatible initial states. We erase state annotations from some of the actions in ξ and η in cases when the corresponding actions in τ model calls from $\mathcal{P}$ to $\mathcal{L}$, which, unlike direct calls from the environment of $\mathcal{P}(\mathcal{L})$ to $\mathcal{L}$, are not annotated with states.

Lemma 12 (Decomposition). *Assume* $\Gamma_0 \vdash \mathcal{L} : \Gamma_1$ *and* $\Gamma_1 \vdash \mathcal{P} : \Gamma_2$ *safe for* $\mathcal{I}_1$ *and* $\mathcal{I}_2$, *respectively. Then*

$$\begin{aligned}&\forall (\theta, l, \tau) \in [\![(\Gamma_0 \vdash \mathcal{P}(\mathcal{L}) : \Gamma_2), \mathcal{I}_2 \otimes \mathcal{I}_1]\!].\ \exists (\theta_1, l_1, \xi) \in [\![\mathcal{L}, \mathcal{I}_1]\!].\\ &\exists (\theta_2, l_2, \eta) \in [\![\mathcal{P}, \mathcal{I}_2]\!].\ (\theta, l, \tau) \in (\theta_1, l_1, \xi) \circ_{\Gamma_0,\Gamma_1,\Gamma_2} (\theta_2, l_2, \eta).\end{aligned}$$

We say that a history H *respects the atomicity of* Γ if for every method declared atomic in Γ, a call to it in H can only be immediately followed by its return. For H and τ such that $\mathsf{history}_M(\tau) = H$, we say that τ *respects the atomicity of* H if τ does not insert an action between adjacent calls and returns by the same thread in H.

Lemma 13 (Rearrangement). *Consider* $\Gamma \vdash \mathcal{P} : \Gamma'$ *and histories* H_1 *and* H_2 *over the set of methods* $\mathsf{dom}(\Gamma)$ *such that* $H_1 \sqsubseteq H_2$. *Assume* H_2 *respects the atomicity of* Γ *and* H_1 *and* H_2 *are well-balanced from their respective initial footprints* l_1 *and* l_2 *such that* $l_1 \succeq l_2$. *If* $\mathcal{P}$ *is safe at* (θ_1, l_1) *for a state* θ_1, *then*

$$\begin{aligned}&\forall \eta_1.\ (\theta_1, l_1, \eta_1) \in [\![\mathcal{P}]\!] \wedge \mathsf{history}_{\mathsf{dom}(\Gamma)}(\eta_1) = H_1 \Rightarrow\\ &\exists \eta_2.\ (\theta_1, l_2, \eta_2) \in [\![\mathcal{P}]\!] \wedge \mathsf{history}_{\mathsf{dom}(\Gamma)}(\eta_2) = H_2\ \wedge\\ &\qquad \eta_1 \lhd_{\mathsf{dom}(\Gamma)} \eta_2\ \wedge\ \eta_2 \text{ respects the atomicity of } H_2.\end{aligned}$$

Lemma 14 (Composition). *Assume* $\Gamma_0 \vdash \mathcal{L} : \Gamma_1$ *and* $\Gamma_1 \vdash \mathcal{P} : \Gamma_2$ *safe for* $\mathcal{I}_1$ *and* $\mathcal{I}_2$, *respectively. Then*

$$\begin{aligned}&\forall (\theta_1, l_1, \xi) \in [\![\mathcal{L}, \mathcal{I}_1]\!].\ \forall (\theta_2, l_2, \eta) \in [\![\mathcal{P}, \mathcal{I}_2]\!].\\ &((\theta_2, l_2) \otimes (\theta_1, l_1))\downarrow \wedge\ \mathsf{history}_{\mathsf{dom}(\Gamma_1)}(\xi) = \mathsf{history}_{\mathsf{dom}(\Gamma_1)}(\eta) \Rightarrow\\ &\qquad \exists (\theta, l, \tau) \in [\![(\Gamma_0 \vdash \mathcal{P}(\mathcal{L}) : \Gamma_2), \mathcal{I}_2 \otimes \mathcal{I}_1]\!].\\ &\qquad (\theta, l, \tau) \in (\theta_1, l_1, \xi) \circ_{\Gamma_0,\Gamma_1,\Gamma_2} (\theta_2, l_2, \eta).\end{aligned}$$

Proof of Theorem 9. Take $(\theta_1, l_1, \tau_1) \in [\![\mathcal{P}(\mathcal{L}_1), \mathcal{I} \otimes \mathcal{I}_1]\!]$. Let $M = \mathsf{dom}(\Gamma_1)$ and $M' = (\mathsf{dom}(\Gamma_1) - \mathsf{dom}(\Gamma_0)) \cap \mathsf{dom}(\Gamma_2)$. By

Lemma 12, for some ξ_1, η_1, θ_1' and θ we have

$$(\theta_1', l_1, \xi_1) \in [\![\mathcal{L}_1, \mathcal{I}_1]\!] \wedge (\theta, \delta(\theta_1') * l_1, \eta_1) \in [\![\Gamma_1 \vdash \mathcal{P} : \Gamma_2, \mathcal{I}]\!] \\ \wedge \theta_1 = \theta * \theta_1' \wedge \mathsf{history}_M(\xi_1) = \mathsf{history}_M(\eta_1) \\ \wedge \mathsf{client}_M(\tau_1) = \mathsf{ierase}_{M'}(\eta_1). \quad (21)$$

Let $H_1 = \mathsf{history}_M(\xi_1)$. Since $(\mathcal{L}_1, \mathcal{I}_1) \sqsubseteq (\mathcal{L}_2, \mathcal{I}_2)$, there exist $(\theta_2', l_2, H_2) \in \mathsf{history}_M(\mathcal{L}_2, \mathcal{I}_2)$ such that

$$H_1 \sqsubseteq H_2 \wedge l_1 \succeq l_2 \wedge \theta_1' \succeq \theta_2'.$$

By Proposition 7, H_1 and H_2 are well-balanced from $\delta(\theta_1') * l_1$ and $\delta(\theta_2') * l_2$, respectively. Since $\Gamma_i \sqsubseteq \Gamma_i'$, H_2 respects the atomicity of Γ_1. Applying Lemma 13, we can construct a trace η_2 such that

$$(\theta, \delta(\theta_2') * l_2, \eta_2) \in [\![\Gamma_1 \vdash \mathcal{P} : \Gamma_2]\!] \wedge \mathsf{history}_M(\eta_2) = H_2 \wedge \eta_1 \lhd_M \eta_2$$

and η_2 respects the atomicity of H_2. The latter implies $(\theta, \delta(\theta_2') * l_2, \eta_2) \in [\![\Gamma_1' \vdash \mathcal{P} : \Gamma_2']\!]$. From (20) we get $(\theta, \delta(\theta_2') * l_2) \in \mathcal{I}$. Since $(\theta_2', l_2, H_2) \in \mathsf{history}(\mathcal{L}_2, \mathcal{I}_2)$, there exists a library trace ξ_2 such that $(\theta_2', l_2, \xi_2) \in [\![\mathcal{L}_2, \mathcal{I}_2]\!]$ and $\mathsf{history}_M(\xi_2) = H_2$. Besides, $((\theta, \delta(\theta_2') * l_2) \otimes (\theta_2', l_2))\downarrow$. Then by Lemma 14, for some trace τ_2, we have $(\theta * \theta_2', l_2, \tau_2) \in [\![\Gamma_0 \vdash \mathcal{P}(\mathcal{L}_2) : \Gamma_2', \mathcal{I} \otimes \mathcal{I}_2]\!]$ and $\mathsf{ierase}_{M'}(\eta_2) = \mathsf{client}_M(\tau_2)$. Since $\eta_1 \lhd_M \eta_2$, we have $\mathsf{ierase}_{M'}(\eta_1) \lhd_M \mathsf{ierase}_{M'}(\eta_2)$. Thus, $\mathsf{ierase}_{M'}(\eta_1) \lhd_M \mathsf{client}_M(\tau_2)$, which, together with (21), implies $\mathsf{client}_M(\tau_1) \lhd_M \mathsf{client}_M(\tau_2)$. □

Discussion. We now highlight some of the technical challenges arising in the proofs of Lemmas 12–14 and Theorem 9.

Most of the proof of the Decomposition Lemma (Lemma 12) deals with maintaining a splitting of the state of $\mathcal{P}(\mathcal{L})$ into the parts owned by $\mathcal{L}$ and $\mathcal{P}$, which changes during ownership transfers. The resulting partial states then define the executions of ξ and η, showing that the traces indeed belong to the semantics of $\mathcal{L}$ and $\mathcal{P}$, respectively. Conversely, the Composition Lemma (Lemma 14) composes the states of $\mathcal{L}$ and $\mathcal{P}$ into a state of $\mathcal{P}(\mathcal{L})$ to construct an execution of τ in the semantics of $\mathcal{P}(\mathcal{L})$.

The Decomposition and the Composition Lemmas rely crucially on the the Strong Locality property (2) of primitive commands, which ensures that their results are independent from parts of the heap they do not access. The property guarantees that client actions are reproducible after we replace one library implementation with another.

Another subtlety in the proofs of the Decomposition and the Composition Lemmas is the need to ensure that ownership transfers to $\mathcal{P}(\mathcal{L})$ from its environment are also reproducible after the above transformation. As we explained in Section 4, the environment respects the notion of ownership, i.e., the states it transfers to $\mathcal{P}(\mathcal{L})$ have to be compatible with the state owned by the program. We then need to ensure that this condition is satisfied when the environment performs transfers to $\mathcal{P}(\mathcal{L}')$ instead. We prove this by tracking the footprint of the imported libraries $\mathcal{L}$ in the semantics of the client $\mathcal{P}$ and restricting the semantics so that the transfers from the environment be compatible with this footprint (rules (15) and (18) in Figure 2). As the abstract library implementation $\mathcal{L}'$ is required to have a smaller footprint than the concrete one $\mathcal{L}$ (see the discussion in Section 4), this ensures that the transfers are still possible after the Composition Lemma plugs in the library $\mathcal{L}'$ into a trace of the client $\mathcal{P}$.

The Rearrangement Lemma (Lemma 13) is the most difficult one of the three. Its proof transforms η_1 into η_2 by repeatedly swapping adjacent actions according to a certain strategy to make the history of methods in $\mathsf{dom}(\Gamma)$ equal to H'. The challenges include devising an appropriate strategy, making sure the intermediate traces are derivable in the semantics of the client $\mathcal{P}$, and, in particular, establishing that all ownership transfers can still be performed after the transformations.

For example, one transformation we have to perform on the trace of the client $\mathcal{P}$ is swapping actions in the trace fragment $(t_1, \mathsf{call}\ m_1(\theta_1))\ (t_2, c)$, where $t_1 \neq t_2$ and c is a primitive command executed by the client. Informally, we justify the validity of this transformation as follows. According to rule (12), the call to m_1 gives up the ownership of the part of the client state θ_1 satisfying its precondition. The safety of the client $\mathcal{P}$ ensures that c does not access the memory transferred. This makes it possible to postpone transferring θ_1 to the library until after c is executed, which ensures that the trace with the two actions swapped belongs to the semantics of $\mathcal{P}$.

It is trickier to swap $(t_1, \mathsf{call}\ m_1(\theta_1))\ (t_2, \mathsf{ret}\ m_2(\theta_2))$, where $t_1 \neq t_2$. The justification of this transformation relies on the fact that the target history H' is well-balanced (Section 4). Consider the case when $\theta_1 = \theta_2 = \theta$. Then the two actions correspond to the client first transferring θ to the library and then getting it back. It is impossible for the library to transfer θ to the client earlier, unless it already owned θ before the call in the original trace. Fortunately, using the fact that H' is well-balanced, we can prove that the latter is indeed the case, and hence, the actions commute.

6. Applications

To demonstrate that the Abstraction Theorem yields an effective method of modularising the verification of concurrent programs, we show how it can be used to verify the linearizability of *non-blocking* concurrent algorithms compositionally in their structure. Non-blocking algorithms employ synchronisation techniques alternative to the usual lock-based mutual exclusion, such as atomic compare-and-swap (CAS) operations provided on modern processors. It is these algorithms that are nowadays used to implement standard concurrent containers in common libraries, such as `java.util.concurrent` and Threading Building Blocks. Non-blocking algorithms are extremely difficult to design and verify. Therefore, it is desirable to split the problem of reasoning about such an algorithm into manageable pieces.

We consider two algorithms, each using a simpler inner algorithm $\mathcal{L}_1$, given by a separate set of methods, to implement a more complicated outer one $\mathcal{L}_2$. In both cases, we prove the linearizability of the whole algorithm $\mathcal{L}_2(\mathcal{L}_1)$ according to Corollary 11, i.e., by showing that:

- the inner algorithm $\mathcal{L}_1$ is linearized by its abstract atomic implementation $\mathcal{L}_1'$;
- the outer algorithm $\mathcal{L}_2$ is linearizable assuming it uses the abstract atomic implementation $\mathcal{L}_1'$ of the inner one.

This decomposes the verification of the algorithm according to the architecture used by its designer. As we argue below, proofs of the algorithms considered using existing methods, without atomicity abstraction, would be at best complicated and at worst, intractable. Due to space constraints we refer the reader to Appendix E for detailed proofs; here we only discuss how our verification method deals with the challenging features of the algorithms.

Non-blocking queue with a memory allocator. As the first example, we consider a well-known implementation of a non-blocking concurrent queue due to Michael and Scott [21], which instantiates the sketch presented in Figure 1 (see Appendix E for the complete source code). The queue algorithm uses a custom memory allocator for nodes in the linked list representing the queue. To avoid the allocator becoming a performance bottleneck, Michael and Scott also implement it using a non-blocking algorithm—a concurrent stack due to Treiber [30]. We verify the example using the above decomposition, i.e., by first linearizing the allocator, and then the queue algorithm using its atomic implementation. We now list the problematic features of the algorithm and justify why such a decomposition is desirable.

First, as we explained in Section 1, the memory allocator and its client transfer the ownership of memory cells at calls to and returns from the former. In particular, for an appropriate definition of the algebra State, the specifications of the allocator methods look approximately as follows:

$\{\mathsf{emp}\}\ \texttt{alloc}()\ \{(\texttt{ret} = 0 \wedge \mathsf{emp}) \vee (\texttt{ret} \neq 0 \wedge \mathsf{Block}(\texttt{ret}))\}$
$\{\mathsf{Block}(\texttt{block})\}\ \texttt{free}(\texttt{block})\ \{\mathsf{emp}\}$

Here `ret` denotes the return value of `alloc`, `block` the value of

the parameter of `free`, emp the empty heap ϵ, and Block(`block`) a block of memory at the address `block` managed by the allocator. The `alloc` method returns 0 when the allocator runs out of memory. Using our logic for proving linearizability (Appendix D), we can show that the allocator with this specification is linearizable with respect to its atomic implementation where the addresses of free memory blocks are stored in an abstract set, instead of a free-list managed by the non-blocking stack algorithm. This requires proving that the ownership transfer is performed correctly.

Second, as we argued in Section 1, when the allocator is accessible to the client of the queue, the combination of the queue algorithm and the allocator might not be linearizable if the client can notice the difference between the memory usage behaviour of the concrete and the abstract queue implementation. To show that this is not the case, while proving the linearizability of the queue implementation using the memory allocator, we need to establish a correspondence between the set of free memory blocks in the concrete and the abstract implementations. After the memory allocator is linearized, referring to the set of free blocks is not problematic, since the allocator state is represented abstractly and is updated atomically. Had we considered the original allocator implementation while proving the queue linearizable, we would have to refer to the internal free-list representation of the allocator to state the required invariant about the queue memory usage. Atomicity abstraction lets us avoid this and makes the proof more modular.

Finally, the parts of the memory owned by the queue and the allocator are not strictly disjoint for the following reason. Non-blocking algorithms often need to check that the information on the basis of which a change to a data structure was computed is still valid when the data structure is updated accordingly. This is usually done by ensuring that certain fields in the data structure have not been changed since the last time the thread making the update read them. The CAS operation allows checking that a field still has the old value atomically with the update. However, this equality does not always imply that the field has not been changed. Namely, a so-called *ABA problem* may arise, when the data structure is changed from its original state *A* to *B*, and then restored to *A* again. For example, a queue node can be returned to the allocator, and then allocated and inserted into the data structure again. To avoid this problem, the queue and the allocator use *modification counters*, atomically incremented at every write to certain memory cells in the data structures, which allows distinguishing between the two *A*s in *ABA*. However, in the case when a validation of the update fails due to a modification counter mismatch, the read of the counter might access a node that is no longer present in the data structure, i.e., has been returned to the allocator in the case of the queue, or has been allocated to the queue in the case of the allocator. Thus, in our proof we cannot consider the state of the allocator as being completely disjoint from the state of its client. Another subtlety is that the queue algorithm relies on the allocator not erasing the modification counters of the node structures it manages, so that the algorithm could still check them after nodes are deallocated. In our proof, we deal with these problems using permissions (Section 2): the queue and the allocator share the right to access some memory cells in restricted ways.

Multiple-word compare-and-swap (MCAS). Vafeiadis [32, Section 5.3] proved the linearizability of the implementation of a multiple-word compare-and-swap (MCAS) operation in terms of single-word CASes by Harris et al. [15]. The MCAS algorithm is extremely complicated; to simplify its structure, the authors singled out a part of it as an auxiliary linearizable operation implementing a restricted version of a two-word comparison and a single-word swap (RDCSS). Clients of both MCAS and RDCSS are expected to transfer the ownership of dynamically-allocated descriptors identifying the operations being performed to the libraries.

Vafeiadis's proof of the linearizability of MCAS is done according to the decomposition presented at the beginning of this section, i.e., by first proving the RDCSS linearizable and then proving the linearizability of MCAS assuming an atomic implementation of RDCSS. Our Abstraction Theorem shows that such a compositional proof is indeed legal, which was not justified by Vafeiadis. Moreover, performing an atomicity abstraction in this case is crucial for the proof to be tractable. Proving linearizability of a library method is typically done by identifying a *linearization point* in its code, where, informally, the method "takes effect" (see Appendix D). After applying the atomicity abstraction to the inner RDCSS algorithm, the linearization point for one of the methods of MCAS is precisely the atomic RDCSS operation. If we considered the original non-atomic RDCSS implementation instead, finding the linearization point would be extremely difficult and would involve reasoning about both MCAS and RDCSS implementations at the same time. In this case, atomicity abstraction helps simplify identifying linearization points (in fact, creating them!) in a complex concurrent algorithm.

7. Related work

The most closely related work is that by Gotsman and Yang [14], who have recently proposed a generalisation of linearizability to deal with liveness properties and proved a corresponding Abstraction Theorem. Since the main focus of that paper was on liveness, it assumed a simplistic setting, where a single library and its client reside in separate address spaces; thus, they are guaranteed not to interfere with each other and cannot perform ownership transfers. As we have shown, lifting these restrictions is non-trivial and is precisely the subject of our main technical contributions. These include the definition of linearizability in a realistic setting (Section 4), a novel notion of client healthiness in the presence of ownership transfer (Section 3), and, last but not least, the proof of the Abstraction Theorem (Section 5). We note that the main challenges in proving the Abstraction Theorem in a realistic setting are also due to ownership transfers (see the discussion in Section 5.2).

The above-mentioned result of Gotsman and Yang's built on a work by Filipović et al. [12], which characterised linearizability in terms of observational refinement over a highly idealistic semantics. The assumptions about the isolation of the library and client made in that work were similar to the ones used by Gotsman and Yang. Besides, Filipović et al. did not justify any compositional proof methods, as we have done in Theorem 9.

Turon and Wand [31] have proposed a logic for establishing refinements between concurrent modules, in fact 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. [9, 10] 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. is non-modular: it 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.

As we noted in Section 1, it is also possible to decompose the verification of concurrent programs into verifying separate threads with the aid of thread-modular techniques [19, 27]. We positioned our approach as performing intrathread-modular reasoning, which goes further by decomposing the verification of code *inside* threads. In this comparison, however, we ignored a restricted

form of intrathread-modular reasoning enabled by thread-modular techniques, which we are now in a position to discuss. Thread-modular methods allow reasoning about the control of a thread in a program while ignoring the possibility of its interruption by the other threads. Hence, they allow considering a library method called by the thread in isolation, e.g., by using the standard proof rules for procedures. However, such a decomposition is done under fixed assumptions on the environment of the thread and thus does not allow, e.g., increasing the atomicity of its actions. As the example of MCAS shows (Section 6), this is necessary to deal with complex algorithms. In the case of MCAS, our method is able to abstract RDCSS to its atomic specification in *all* threads at once. Thus, when reasoning about the transformed MCAS algorithm using our thread-modular logic for linearizability, we can rely on RDCSS actions performed by the environment of a thread to be atomic.

Jacobs et al. [18] have proposed a method for achieving procedural abstraction in the presence of concurrency. They also aim at achieving intrathread-modular reasoning, but in a standard program logic verifying a given program. On the other hand, our focus is on reasoning principles for relating pairs of programs, which use different data structures and meet different atomicity conditions.

Ways of establishing relationships between different sequential implementations of the same library have been studied in *data refinement* [17, 28], including cases of interactions via ownership transfer [3, 13, 22]. Our results can be viewed as generalising data refinement to the concurrent setting. Moreover, when specialised to the sequential case, they provide a more flexible method of performing it in the presence of the heap and ownership transfer than previously proposed ones. In more detail, the way we define client healthiness (Section 3) is more general than the one often used in data refinement [13]. There, it is typical to fix a (precise) invariant of a library and check that the client does not access the area of memory fenced off by the invariant. Here we do not require an explicit library invariant, using client-local semantics instead (Section 3): since primitive commands fault when accessing non-existent memory cells, the safety of the client in this semantics ensures that it does not access the internals of the library. We note that the approach requiring an invariant for library-local data structures does not generalise to the concurrent setting: while a precise invariant for the data structures *shared* among threads executing library code is not usually difficult to find, the state of data structures *local* to the threads depends on their program counters. Thus, an invariant insensitive to program positions inside the library code often does not exist. Such difficulties are one of reasons for using client- and library-local semantics in this paper.

If Theorem 9 generalises data refinement to concurrent setting, then Theorem 3 does the same for information hiding [26]. The latter is concerned with compositionally verifying a (single) program consisting of a library and its client executing in a shared address space; data refinement can be viewed as its relational version. Theorem 3 is a concurrent analogue of the procedure declaration rule with information hiding proposed by O'Hearn et al. [25]. As in the case of data refinement, the conditions required by the theorem are somewhat more general than in existing proof systems [11, 23, 25].

Finally, we note that the applicability of our results is not limited to proving existing programs correct: they can also be used in the context of formal program development. In this case, instead of *abstracting* an existing library to an atomic specification while proving a complete program, Theorem 9 allows *refining* an atomic library specification to a concrete concurrent implementation while developing a program top-down [2, 20]. Our work thus advances the method of atomicity refinement to a setting with concurrent components sharing an address space and communicating via ownership transfers.

8. Conclusion

The only way to prove modern concurrent systems correct is by decomposing them into manageable pieces according to their architecture. This is challenging because software components are not isolated, but interact with their environment in complicated ways. In this paper we have shown that, even for subtle interactions involving the transfer of ownership of memory areas and access to a shared state, it is possible to separate the verification of concurrent libraries from that of their clients. We consider this work a starting point for developing the theory of abstractions for different types of interactions among concurrent components that would allow such decompositions. Of particular interest for the future work are ways of verifying recursive libraries compositionally and handling interactions arising from a concurrent program running on a weak memory model.

References

[1] D. Amit, N. Rinetzky, T. W. Reps, M. Sagiv, and E. Yahav. Comparison under abstraction for verifying linearizability. In *CAV*, 2007.

[2] R.-J. Back. On correct refinement of programs. *JCSS*, 1981.

[3] A. Banerjee and D. A. Naumann. Ownership confinement ensures representation independence in object-oriented programs. *JACM*, 2005.

[4] R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permission accounting in separation logic. In *POPL*, 2005.

[5] C. Boyapati, R. Lee, and M. C. Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *OOPSLA*, 2002.

[6] C. Calcagno, P. O'Hearn, and H. Yang. Local action and abstract separation logic. In *LICS*, 2007.

[7] D. G. Clarke, J. Noble, and J. M. Potter. Simple ownership types for object containment. In *ECOOP*, 2001.

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

[9] T. Elmas, S. Qadeer, and S. Tasiran. A calculus of atomic actions. In *POPL*, 2009.

[10] T. Elmas, S. Qadeer, A. Sezgin, O. Subasi, and S. Tasiran. Simplifying linearizability proofs with reduction and abstraction. In *TACAS*, 2010.

[11] X. Feng. Local rely-guarantee reasoning. In *POPL*, 2009.

[12] I. Filipović, P. O'Hearn, N. Rinetzky, and H. Yang. Abstraction for concurrent objects. In *TCS*, 2010.

[13] I. Filipović, P. O'Hearn, N. Torp-Smith, and H. Yang. Blamining the client: On data refinement in the presence of pointers. *FAC*, 2010.

[14] A. Gotsman and H. Yang. Liveness-preserving atomicity abstraction. In *ICALP*, 2011.

[15] T. Harris, K. Fraser, and I. Pratt. A practical multi-word compare-and-swap operation. In *DISC*, 2002.

[16] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *TOPLAS*, 1990.

[17] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1972.

[18] B. Jacobs and F. Piessens. Expressive modular fine-grained concurrency specification. In *POPL*, 2011.

[19] C. B. Jones. Specification and design of (parallel) programs. In *IFIP Congress*, 1983.

[20] C. B. Jones. Splitting atoms safely. *TCS*, 2007.

[21] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC*, 1996.

[22] I. Mijajlovic and H. Yang. Data refinement with low-level pointer operations. In *APLAS*, 2005.

[23] D. A. Naumann and A. Banerjee. Dynamic boundaries: Information hiding by second order framing with first order assertions. In *ESOP*, 2010.

[24] P. O'Hearn. Resources, concurrency and local reasoning. *TCS*, 2007.

[25] P. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *POPL*, 2004.

[26] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *CACM*, 1972.

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

[28] J. C. Reynolds. Types, abstraction and parametric polymorphism. In *IFIP Congress*, 1983.

[29] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, 2002.

[30] R. K. Treiber. Systems programming: Coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center, 1986.

[31] A. Turon and M. Wand. A separation logic for refining concurrent objects. In *POPL*, 2011.

[32] V. Vafeiadis. Modular fine-grained concurrency verification. PhD Thesis. University of Cambridge, 2008.

A. Additional definitions for the programming language

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

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

where expressions E are defined as follows:

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

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 evalues to 0, and 0 otherwise. We denote with $[\![E]\!]_{\theta,t} \in \mathsf{Val} \cup \{\top\}$ the result of evaluating the expression E in the state θ with the current thread identifier t. This evaluation might dereference illegal memory adresses, such as dangling pointers, and 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 3. Using this transition relation, we then define $f_c^t : \mathsf{RAM} \to \mathcal{P}(\mathsf{RAM})^\top$ for the primitive commands c as follows:

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

Definition of loops and conditionals. The standard commands for conditionals and loops are defined in our language as follows:

$$(\mathsf{if}\ E\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2) = (\mathsf{assume}(E); C_1) + (\mathsf{assume}(!E); C_2),$$
$$(\mathsf{while}\ E\ \mathsf{do}\ C) = (\mathsf{assume}(E); C)^*; \mathsf{assume}(!E).$$

Translation of commands to CFGs. We construct the CFG of a command C by induction on its syntax:

1. A primitive command c has the CFG
$$(\{\mathsf{start}, \mathsf{end}\}, \{(\mathsf{start}, c, \mathsf{end})\}, \mathsf{start}, \mathsf{end}).$$
2. Assume C_1 and C_2 have CFGs $(N_1, T_1, \mathsf{start}_1, \mathsf{end}_1)$ and $(N_2, T_2, \mathsf{start}_2, \mathsf{end}_2)$, respectively. Then $C_1; C_2$ has the CFG
$$(N_1 \cup N_2, T_1 \cup T_2 \cup \{(\mathsf{end}_1, \mathsf{skip}, \mathsf{start}_2)\}, \mathsf{start}_1, \mathsf{end}_2).$$
3. Assume C_1 and C_2 have CFGs $(N_1, T_1, \mathsf{start}_1, \mathsf{end}_1)$ and $(N_2, T_2, \mathsf{start}_2, \mathsf{end}_2)$, respectively. Then $C_1 + C_2$ has the CFG
$$(N_1 \cup N_2 \cup \{\mathsf{start}, \mathsf{end}\}, T_1 \cup T_2 \cup \{(\mathsf{start}, \mathsf{skip}, \mathsf{start}_1), (\mathsf{start}, \mathsf{skip}, \mathsf{start}_2), (\mathsf{end}_1, \mathsf{skip}, \mathsf{end}), (\mathsf{end}_2, \mathsf{skip}, \mathsf{end})\}, \mathsf{start}, \mathsf{end}).$$
4. Assume C has a CFG $(N, T, \mathsf{start}, \mathsf{end})$. Then C^* has the CFG
$$(N, T \cup \{(\mathsf{end}, \mathsf{skip}, \mathsf{start})\}, \mathsf{start}, \mathsf{end}).$$

Example of a separation algebra with permissions. We now present an extension of the separation algebra RAM in Section 2, where states carry additional information regarding permission 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. In Section E we give a more complicated algebra used for proving one of our examples. Formally, this algebra, denoted RAM_p, is defined as follows:

$$\mathsf{Loc} = \mathbb{N}^+ \qquad \mathsf{Val} = \mathbb{Z} \qquad \mathsf{Perm} = (0, 1]$$
$$\mathsf{RAM}_p = \mathsf{Loc} \rightharpoonup_{fin} (\mathsf{Val} \times \mathsf{Perm})$$

A state in this model consists of a finite parial 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 [4]. As we show below, 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 [8].

Figure 3. Transition relation for sample primitive commands in the RAM model. The result $\top$ indicates that the command faults.

(skip, θ)	$\rightsquigarrow_t \theta$	
$[E]{=}E', \theta$	$\rightsquigarrow_t \theta[[\![E]\!]_{\theta,t} : [\![E']\!]_{\theta,t}]$	if $[\![E]\!]_{\theta,t}, [\![E']\!]_{\theta,t} \in \mathsf{dom}(\theta)$
$[E]{=}E', \theta$	$\rightsquigarrow_t \top$	if $\neg([\![E]\!]_{\theta,t}, [\![E']\!]_{\theta,t} \in \mathsf{dom}(\theta))$
$\mathsf{assume}(E), \theta$	$\rightsquigarrow_t \theta$	if $[\![E]\!]_{\theta,t} \in \mathsf{Val},\ [\![E]\!]_{\theta,t} \neq 0$
$\mathsf{assume}(E), \theta$	$\rightsquigarrow_t \top$	if $[\![E]\!]_{\theta,t} = \top$

Figure 4. Transition relation for sample primitive commands in the RAM_p model. The evaluation of expressions $[\![E]\!]$ ignores the permission part of the model.

skip, θ	$\rightsquigarrow_t \theta$	
$[E]{=}E', \theta$	$\rightsquigarrow_t \theta[[\![E]\!]_{\theta,t} : ([\![E']\!]_{\theta,t}, 1)]$	if $\theta([\![E]\!]_{\theta,t}) = (\_, 1),\ [\![E']\!]_{\theta,t} \in \mathsf{Val}$
$[E]{=}E', \theta$	$\rightsquigarrow_t \top$	if the above condition does not hold
$\mathsf{assume}(E), \theta$	$\rightsquigarrow_t \theta$	if $[\![E]\!]_{\theta,t} \in \mathsf{Val},\ [\![E]\!]_{\theta,t} \neq 0$
$\mathsf{assume}(E), \theta$	$\rightsquigarrow_t \top$	if $[\![E]\!]_{\theta,t} = \top$

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:

$$\forall x \in \mathsf{Loc}.\, \theta_1(x)\downarrow \wedge \theta_2(x)\downarrow \Rightarrow (\exists v, \pi_1, \pi_2.\, \theta_1(x) = (v, \pi_1) \wedge \theta_2(x) = (v, \pi_2) \wedge \pi_1 + \pi_2 \leq 1).$$

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

$$\theta_1 * \theta_2 = \{(x, (v, \pi)) \mid (\theta_1(x) = (v, \pi) \wedge \theta_2(x)\uparrow) \vee (\theta_2(x) = (v, \pi) \wedge \theta_1(x)\uparrow) \vee (\theta_1(x) = (v, \pi_1) \wedge \theta_2(x) = (v, \pi_2) \wedge \pi = \pi_1 + \pi_2)\}.$$

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

$$\delta(\theta) = \{\theta' \mid \forall x.\, (\theta(x)\downarrow \Leftrightarrow \theta'(x)\downarrow) \wedge \forall v, \pi.\, (\theta(x) = (v, \pi) \vee \theta'(x) = (v, \pi)) \wedge \pi < 1 \Rightarrow \theta(x) = \theta'(x)\}.$$

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 conditions in Definition 2 are satisfied.

Finally, we define $f_c^t : \mathsf{RAM}_p \to \mathcal{P}(\mathsf{RAM}_p)^\top$ for the primitive commands c, following the same recipe as in the RAM case. In this case, we use the transition relation described in Figure 4.

An additional property of $*$ and $\setminus$. In our proofs we rely on the following associativity property of $*$ and $\setminus$, which needs to be checked for a particular instatiation of State with a separation algebra. Let $x, y, l_1, \ldots, l_n \in \mathsf{Foot}$. Consider two defined expressions $E_1(x)$ and $E_2(x * y)$, applying a sequence of $*$ and $\setminus$ operations to x, respectively, $x * y$: e.g., $x * l_{i_1} \setminus l_{i_2} * l_{i_3} \ldots$. Let us assume that *(i)* there is a unique occurence of every l_i in E_j; *(ii)* each l_i is used with the same operation in E_1 and E_2; *(iii)* every l_i used in E_1 with $*$ also occurs in E_2; *(iv)* the sets of l_i used in E_1 and E_2 with $\setminus$ are the same. Then $E_2(x * y) \setminus y$ is defined.

This property is used in the proof of our Rearrangement Lemma (Section B.4). Checking it for typical models of program states is not problematic. However, we conjecture that, in the context we use it, we should be able to derive it from more primitive axioms. We plan to investigate this.

B. Proofs

We first define several operations that relate configurations in the semantics of $\mathcal{P}(\mathcal{L})$, $\mathcal{P}$ and $\mathcal{L}$. When considering the CFG of $\mathcal{P}$ and $\mathcal{L}$, we disambiguate the nodes v^t_{mgc} in $\mathcal{P}$ and $\mathcal{L}$ by denoting the latter ones with w^t_{mgc} and the former with v^t_{mgc}. Also, we remind the

reader that when a method m is not implemented in $\mathcal{P}$ but called there, we introduced a particular node v_m for the method m in the semantics of $\mathcal{P}$.

A partial operation $\otimes : \mathsf{Pos} \times \mathsf{Pos} \rightharpoonup \mathsf{Pos}$ combines thread positions in the semantics of $\mathcal{P}$ and $\mathcal{L}$ to obtain a position of $\mathcal{P}(\mathcal{L})$ as follows: $\alpha \otimes v^t_{\mathsf{mgc}} = \alpha$ and $\alpha v_m \otimes v^t_{\mathsf{mgc}}\beta = \alpha\beta$ for $t \in \mathsf{ThreadID}$. For all the other combinations, the $\otimes$ operator is undefined. We lift $\otimes$ to program counters pointwise. We now define a partial operation

$$\otimes : \mathcal{P}(\mathsf{ThreadID})^+ \times \mathcal{P}(\mathsf{ThreadID})^+ \rightharpoonup \mathcal{P}(\mathsf{ThreadID})^+$$

on scheduling policies:

$$\begin{gathered}\mathsf{ThreadID} \otimes \mathsf{ThreadID} = \mathsf{ThreadID},\\ \mathsf{ThreadID} \otimes \mathsf{ThreadID}\,\kappa = \mathsf{ThreadID}\,\kappa,\\ \mathsf{ThreadID}\,\kappa_1 K \otimes \mathsf{ThreadID}\,K\kappa_2 = \mathsf{ThreadID}\,\kappa_1 K\kappa_2;\end{gathered}$$

all other combinations are undefined. We then define the $*$ operation that combines non-erroneous configurations from $\mathsf{PConfig}$ as follows:

$$(\alpha_1, \theta_1, l_1) * (\alpha_2, \theta_2, l_2) = (\alpha_1 \otimes \alpha_2, (\theta_1, l_1) \otimes (\theta_2, l_2)).$$

We also define $*$ on non-erroneous configurations from Config:

$$\begin{aligned}(\mathsf{pc}_1, \theta_1, l_1, \kappa_1) * (\mathsf{pc}_2, \theta_2, l_2, \kappa_2) = \\ (\mathsf{pc}_1 \otimes \mathsf{pc}_2, (\theta_1, l_1) \otimes (\theta_2, l_2), \kappa_1 \otimes \kappa_2).\end{aligned}$$

For a set of methods M we define functions $\mathsf{client}_M : \mathsf{Pos} \to \mathsf{Pos}$ and $\mathsf{lib}^t_M : \mathsf{Pos} \to \mathsf{Pos}$, $t \in \mathsf{ThreadID}$ that compute thread positions in the semantics of $\mathcal{P}$ and $\mathcal{L}$ corresponding to a position in $\mathcal{P}(\mathcal{L})$. First, we let $\mathsf{client}_M(w^t_{\mathsf{mgc}}) = w^t_{\mathsf{mgc}}$ and $\mathsf{lib}_M(w^t_{\mathsf{mgc}}) = v^t_{\mathsf{mgc}}$. Consider now a thread position $\alpha_1\alpha_2$, where α_1 is its maximal prefix consisting only of nodes outside the bodies of methods in M. We let

$$\mathsf{client}_M(\alpha_1\alpha_2) = \alpha_1 v_m,$$

if α_2 is non-empty and m is the method whose implementation contains the first node in α_2. Otherwise, we let $\mathsf{client}_M(\alpha_1) = \alpha_1$. We also let $\mathsf{lib}^t_M(\alpha_1\alpha_2) = v^t_{\mathsf{mgc}}\alpha_2$. We then lift these operations to program counters as follows: $(\mathsf{client}_M(\mathsf{pc}))(t) = \mathsf{client}_M(\mathsf{pc}(t))$ and $(\mathsf{lib}_M(\mathsf{pc}))(t) = \mathsf{lib}_M(\mathsf{pc}(t))$.

Assume a program $\mathcal{P}(\mathcal{L})$, such that $\Gamma_0 \vdash \mathcal{L} : \Gamma_1$ and $\Gamma_1 \vdash \mathcal{P} : \Gamma_2$. Consider a thread $t \in \mathsf{ThreadID}$, a set of methods M, a scheduling policy κ and a thread position $\alpha_1\alpha_2$, where α_1 is its maximal prefix consisting only of nodes outside the bodies of methods in M. Let i_j be the number of nodes in α_j, $j = \{1, 2\}$ belonging to the CFGs of methods declared as atomic in Γ_0, $\mathcal{P}$ or $\mathcal{L}$. We then define $\mathsf{client}_{M,\alpha,t}(\kappa) = \mathsf{ThreadID}\,\kappa_1$ and $\mathsf{lib}_{M,\alpha,t}(\kappa) = \mathsf{ThreadID}\,\kappa_2$, where κ_2 consists of i_2 singleton sets $\{t\}$; κ_1 consists of i_1 singleton sets $\{t\}$, if $i_2 = 0$, and $i_1 + 1$ such sets, otherwise.

For $\varsigma = (\mathsf{pc}[t : \alpha], \theta, l, \kappa) \in \mathsf{Config}$, such that $\mathsf{pc}(t)$ is undefined, we let $\varsigma|_t = (\alpha, \theta, l) \in \mathsf{PConfig}$. For $\sigma = (\alpha, \theta, l) \in \mathsf{PConfig}$, $\mathsf{pc} \in \mathsf{ThreadID} \rightharpoonup_{fin} \mathsf{Pos}$, $\kappa \in \mathcal{P}(\mathsf{ThreadID})^+$ we let $\mathsf{combine}_t(\sigma, \mathsf{pc}, \kappa) = (\mathsf{pc}[t : \alpha], \theta, l, \kappa)$.

B.1 Properties of the $\setminus$ operation

Proposition 15. *The $\setminus$ operation is well-defined.*

Proof. Consider $l_1, l_2 \in \mathsf{Foot}$ and $\theta_1, \theta_2, \theta, \theta'_1, \theta'_2, \theta'$ 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 show that $\delta(\theta) = \delta(\theta')$.

We have:

$$\delta(\theta_1 * \theta) = \delta(\theta_2) = l_2 = \delta(\theta'_2) = \delta(\theta'_1 * \theta').$$

By Property 2 in Definition 2, this implies $\delta(\theta) = \delta(\theta')$. □

We now list some the useful properties of $\setminus$.

Proposition 16. *For all $l_1, l_2, l_3 \in \mathsf{Foot}$, in the case when all subexpressions used in the corresponding case below are defined, we have:*
1. $(l_1 \setminus l_2) * l_2 = l_1$;
2. $(l_1 * l_2) \setminus l_2 = l_1$;
3. $(l_1 * l_2) \setminus l_3 = (l_1 \setminus l_3) * l_2$.

Proof. We consider each property in turn.

1. Consider $\theta \in l_1 \setminus l_2$ and $\theta_2 \in l_2$. Then there exist $\theta_1 \in l_1$ and $\theta'_2 \in l_2$ such that $\theta * \theta'_2 = \theta_1$. By Property 1 in Definition 2 this implies

$$(l_1 \setminus l_2) * l_2 = \delta(\theta) * \delta(\theta'_2) = \delta(\theta * \theta'_2) = \delta(\theta_1) = l_1.$$

2. Consider $\theta \in (l_1 * l_2) \setminus l_2$. Then for some $\theta_1 \in l_1$ and $\theta_2, \theta'_2 \in l_2$ we have $\theta_1 * \theta_2 = \theta'_2 * \theta$. This implies $\delta(\theta_1 * \theta_2) = \delta(\theta'_2 * \theta)$. By Property 1 in Definition 2, this implies

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

3. By item 1 we get

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

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

B.2 Properties of the ownership-transfer semantics

The following proposition, used in later proofs, shows that in any reachable configuration of a program, the footprint of imported libraries is compatible with that of the program.

Proposition 17 (Consistency). *If $\Gamma \vdash \mathcal{P} : \Gamma'$ is safe at (θ_0, l_0), where $\{\theta_0\} * l_0 \neq \emptyset$, and $(\mathsf{pc}_0, \theta_0, l_0, \kappa_0) \longrightarrow^*_{\Gamma,\mathcal{P},\Gamma'} (\mathsf{pc}, \theta, l, \kappa)$, then $\{\theta\} * l \neq \emptyset$.*

Proof. We prove the statement of the proposition by induction on the length of the derivation. The base case holds because $\{\theta_0\} * l_0 \neq \emptyset$. Assume

$$(\mathsf{pc}_0, \theta_0, l_0, \kappa_0) \longrightarrow^*_{\mathcal{P}} (\mathsf{pc}, \theta, l, \kappa) \xrightarrow{\varphi}_{\mathcal{P}} (\mathsf{pc}', \theta', l', \kappa')$$

and $\{\theta\} * l \neq \emptyset$.

We consider every rule from rules (8)–(19) in Figure 2 that can be used to derive φ and can affect the program memory and the footprint of imported libraries:

- Rule (8). For some $c \in \mathsf{PComm}$, $\theta' \in f_c(\theta) \neq \top$ and $l' = l$. Since $\{\theta\} * l \neq \emptyset$, (3) implies that $\{\theta'\} * l' \neq \emptyset$.
- Rule (12). For some predicate p_t and $\theta_p \in p_t$ we have $\theta = \theta' * \theta_p$ and $l' = l * \delta(\theta_p)$. In this case $\{\theta'\} * l' = \{\theta'\} * l * \delta(\theta_p)$, which is non-empty when $\{\theta\} * l$ is.
- Rule (14). For some predicate q_t and $\theta_q \in q_t$ we have $\theta' = \theta * \theta_q$ and $l' = l \setminus \delta(\theta_q)$. Then for some $\theta_L \in l$, $\theta'_L \in l'$ and $\theta'_q \in \delta(\theta_q)$ we have $\theta'_L * \theta'_q = \theta_L$. From $\{\theta\} * l \neq \emptyset$ we get that $\theta * \theta_L = \theta'_L * \theta'_q * \theta$ is defined. Since $\theta'_q \in \delta(\theta_q)$, $\theta'_L * \theta_q * \theta$ is defined as well. Hence, $\emptyset \neq (\{\theta\} * \{\theta_q\}) * (l \setminus \delta(\theta_q)) = \{\theta'\} * l'$.
- Rule (15). For some predicate p_t and $\theta_p \in p_t$ we have $\theta' = \theta * \theta_p$ and $l' = l$. In this case $\{\theta'\} * l' = \{\theta\} * \{\theta_p\} * l$, which is non-empty according to the side condition of rule (15).
- Rule (16). For some predicate q_t and $\theta_q \in q_t$ we have $\theta = \theta' * \theta_q$ and $l' = l$. We have $\{\theta\} * l = \{\theta'\} * \{\theta_q\} * l'$, so $\{\theta'\} * l' \neq \emptyset$.
- Rule (18). For some predicate p_t and $\theta_p \in p_t$ we have $\theta' = \theta$ and $l' = l * \delta(\theta_p)$. In this case $\{\theta'\} * l' = \delta(\theta) * l * \delta(\theta_p)$, which is non-empty according to the side condition of rule (18).
- Rule (19). For some predicate q_t and $\theta_q \in q_t$ we have $\theta' = \theta$ and $l' = l \setminus \delta(\theta_q)$. Then $\{\theta'\} * l' = \{\theta\} * (l \setminus \delta(\theta_q))$, which is non-empty when $\{\theta\} * l$ is.

So, $\{\theta'\} * l' \neq \emptyset$ in all cases, as required. □

Proof of Proposition 7. Consider $\Gamma \vdash \mathcal{L} : \Gamma'$ safe for $\mathcal{I}$ and $(\theta_0, l_0, \xi) \in [\![\mathcal{L}, \mathcal{I}]\!]$. Then there exists a derivation of ξ:

$$(\mathsf{pc}_0, \theta_0, l_0, \mathsf{ThreadID}) \xrightarrow{\xi}{}^*_{\mathcal{L}} (\mathsf{pc}, \theta, l, \kappa).$$

Let $M = \mathsf{dom}(\Gamma')$. We prove

$$\mathsf{run}(\mathsf{history}_M(\xi), \delta(\theta_0) * l_0) = \delta(\theta) * l$$

by induction on the length of the derivation of ξ. By Proposition 17, $\delta(\theta) * l \neq \emptyset$, which implies that $\mathsf{history}_M(\xi)$ is well-balanced from $\delta(\theta_0) * l_0$.

The base case trivially follows from the definition of run. For the induction step, assume the above and

$$(\mathsf{pc}, \theta, l, \kappa) \xrightarrow{\varphi}_{\mathcal{L}} (\mathsf{pc}', \theta', l', \kappa').$$

Let

$$l_1 = \mathsf{run}(\mathsf{history}_M(\xi), \delta(\theta_0) * l_0),$$
$$l_2 = \mathsf{run}(\mathsf{history}_M(\xi\varphi), \delta(\theta_0) * l_0).$$

Then $l_1 = \delta(\theta) * l$. We now make a case-split on which of rules (8)–(19) is used to derive φ.

- Rule (15). Then $\varphi = (t, \mathsf{call}\ m(\theta_p))$, where $\Gamma'(m) = (p, \_)$ and $\theta_p \in p_t$. In this case $\theta' = \theta * \theta_p$, $l' = l$, $l_2 = l_1 * \delta(\theta_p)$[1] and $\delta(\theta) * l * \delta(\theta_p) \neq \emptyset$. Then
$$l_2 = l_1 * \delta(\theta_p) = \delta(\theta) * l * \delta(\theta_p) = \delta(\theta') * l' \neq \emptyset.$$
- Rule (16). Then $\varphi \in (t, \mathsf{ret}\ m(\theta_q))$, where $\Gamma'(m) = (\_, q)$ and $\theta_q \in q_t$. In this case $\theta = \theta' * \theta_q$, $l' = l$ and $l_2 = l_1 \setminus \delta(\theta_q)$. We have
$$l_1 = \delta(\theta) * l = \delta(\theta' * \theta_q) * l = \delta(\theta') * \delta(\theta_q) * l.$$
Hence, $l_1 \setminus \delta(\theta_q)$ is defined and $l_2 = \delta(\theta') * l'$.
- Rule (18). Then $\varphi \in (t, \mathsf{call}\ m(\theta_p))$, where $(\Gamma' \cap \Gamma)(m) = (p, \_)$ and $\theta_p \in p_t$. In this case $\theta' = \theta$, $l' = l * \delta(\theta_p)$, $l_2 = l_1 * \delta(\theta_p)$ and $\delta(\theta) * l * \delta(\theta_p) \neq \emptyset$. We have
$$l_2 = l_1 * \delta(\theta_p) = \delta(\theta) * l * \delta(\theta_p) = \delta(\theta') * l',$$
which is defined.
- Rule (19). Then $\varphi \in (t, \mathsf{ret}\ m(\theta_q))$, where $(\Gamma' \cap \Gamma)(m) = (\_, q)$ and $\theta_q \in q_t$. In this case $\theta' = \theta$, $l' = l \setminus \delta(\theta_p)$ and $l_2 = l_1 \setminus \delta(\theta_p)$. We have $l_2 = l_1 \setminus \delta(\theta_p) = (\delta(\theta) * l) \setminus \delta(\theta_p)$. Since $l \setminus \delta(\theta_p)$ is defined, l_2 is defined as well. Besides, $l_2 = (\delta(\theta) * l) \setminus \delta(\theta_p) = \delta(\theta') * l'$.
- It is easy to check that in all other cases we have $\delta(\theta') * l' = \delta(\theta) * l = l_1 = l_2$.

□

B.3 Proof of Lemma 12

Consider $(\theta_0 * \theta_L, l_0, \tau) \in [\![\mathcal{P}(\mathcal{L})]\!]$ such that $(\theta_L, l_0) \in \mathcal{I}_1$ $(\theta_0, \delta(\theta_L) * l_0) \in \mathcal{I}_2$. We need to construct η and ξ such that $(\theta_0, \delta(\theta_L) * l_0, \eta) \in [\![\mathcal{P}]\!]$, $(\theta_L, l_0, \xi) \in [\![\mathcal{L}]\!]$ and

$$\begin{aligned}&\mathsf{history}_{\mathsf{dom}(\Gamma_1)}(\eta) = \mathsf{history}_{\mathsf{dom}(\Gamma_1)}(\xi)\\ \wedge\ &\mathsf{client}_{\mathsf{dom}(\Gamma_1)}(\tau) = \mathsf{ierase}_{(\mathsf{dom}(\Gamma_1)-\mathsf{dom}(\Gamma_0))\cap\mathsf{dom}(\Gamma_2)}(\eta)\\ \wedge\ &\mathsf{erase}_{\mathsf{dom}(\Gamma_1)-\mathsf{dom}(\Gamma_0)}(\mathsf{lib}_{\mathsf{dom}(\Gamma_1)}(\tau)) = \mathsf{erase}_{\mathsf{dom}(\Gamma_1)-\mathsf{dom}(\Gamma_0)}(\xi)\end{aligned}$$

Let

$$\begin{aligned}&\varsigma^0 = (\mathsf{pc}_0, \theta_0 * \theta_L, l_0, \mathsf{ThreadID})\\ \wedge\ &\varsigma_1^0 = (\mathsf{pc}_0^1, \theta_0, \delta(\theta_L) * l_0, \mathsf{ThreadID})\\ \wedge\ &\varsigma_2^0 = (\mathsf{pc}_0^2, \theta_L, l_0, \mathsf{ThreadID}),\end{aligned}$$

where pc_0, pc_0^1 and pc_0^2 are initial program counters of $\mathcal{P}(\mathcal{L})$, $\mathcal{P}$ and $\mathcal{L}$, respectively, such that $\mathsf{pc}_0 = \mathsf{pc}_0^1 * \mathsf{pc}_0^2$. Then $\varsigma^0 = \varsigma_1^0 * \varsigma_2^0$.

Since $(\theta_0 * \theta_L, l_0, \tau) \in [\![\mathcal{P}(\mathcal{L})]\!]$, there exists a τ-labelled derivation using $\longrightarrow_{\mathcal{P}(\mathcal{L})}$ that starts from $\varsigma^0 \in \mathsf{Config}$. From the derivation of τ, we construct traces η and ξ, together with their derivations using $\longrightarrow_{\mathcal{P}}$ and $\longrightarrow_{\mathcal{L}}$. Our construction first considers every finite prefix τ_i of τ and builds traces η_i and ξ_i in the semantics of $\mathcal{P}$ 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 finite prefix τ_i of τ, traces η_i and ξ_i and configurations $\varsigma, \varsigma_1, \varsigma_2 \in \mathsf{Config}$ such that $\varsigma = \varsigma_1 * \varsigma_2 \neq \top$ and

$$\varsigma^0 \xrightarrow{\tau_i}{}^*_{\mathcal{P}} \varsigma\ \wedge\ \varsigma_1^0 \xrightarrow{\eta_i}{}^*_{\mathcal{P}} \varsigma_1 \wedge\ \varsigma_2^0 \xrightarrow{\xi_i}{}^*_{\mathcal{L}} \varsigma_2;$$

$$\begin{aligned}&\mathsf{history}_{\mathsf{dom}(\Gamma_1)}(\eta_i) = \mathsf{history}_{\mathsf{dom}(\Gamma_1)}(\xi_i)\\ \wedge\ &\mathsf{client}_{\mathsf{dom}(\Gamma_1)}(\tau_i) = \mathsf{ierase}_{(\mathsf{dom}(\Gamma_1)-\mathsf{dom}(\Gamma_0))\cap\mathsf{dom}(\Gamma_2)}(\eta_i)\\ \wedge\ &\mathsf{erase}_{\mathsf{dom}(\Gamma_1)-\mathsf{dom}(\Gamma_0)}(\mathsf{lib}_{\mathsf{dom}(\Gamma_1)}(\tau_i)) = \mathsf{erase}_{\mathsf{dom}(\Gamma_1)-\mathsf{dom}(\Gamma_0)}(\xi_i)\end{aligned}$$

If $\tau = \tau_i\varphi\tau'$ for some action φ and trace τ', and $\varsigma \xrightarrow{\varphi}_{\mathcal{P}(\mathcal{L})} \varsigma'$ for some $\varsigma' \neq \top$, there exist $\varsigma_1', \varsigma_2' \in \mathsf{Config} - \{\top\}$ and extensions η_{i+1} and ξ_{i+1} of η_i and ξ_i and the corresponding derivations such that $\varsigma' = \varsigma_1' * \varsigma_2'$ and

$$\varsigma_1^0 \xrightarrow{\eta_{i+1}}{}^*_{\mathcal{P}} \varsigma_1'\ \wedge\ \varsigma_2^0 \xrightarrow{\xi_{i+1}}{}^*_{\mathcal{L}} \varsigma_2';$$

$$\begin{aligned}&\mathsf{history}_{\mathsf{dom}(\Gamma_1)}(\eta_{i+1}) = \mathsf{history}_{\mathsf{dom}(\Gamma_1)}(\xi_{i+1})\\ \wedge\ &\mathsf{client}_{\mathsf{dom}(\Gamma_1)}(\tau_i\varphi) = \mathsf{ierase}_{(\mathsf{dom}(\Gamma_1)-\mathsf{dom}(\Gamma_0))\cap\mathsf{dom}(\Gamma_2)}(\eta_{i+1})\\ \wedge\ &\mathsf{erase}_{\mathsf{dom}(\Gamma_1)-\mathsf{dom}(\Gamma_0)}(\mathsf{lib}_{\mathsf{dom}(\Gamma_1)}(\tau_{i+1})) = \mathsf{erase}_{\mathsf{dom}(\Gamma_1)-\mathsf{dom}(\Gamma_0)}(\xi_{i+1})\end{aligned}$$

To prove the claim, we assume $\tau_i, \varphi, \tau', \varsigma, \varsigma_1, \varsigma_2, \varsigma', \eta_i, \xi_i$ satisfying the assumptions in it. Let $\varphi = (t, \_)$,

$$\varsigma = (\mathsf{pc}[t : \alpha_0], \theta, l, \kappa)\ \wedge\ \varsigma' = (\mathsf{pc}[t : \alpha_0'], \theta', l', \kappa'),$$

where $\mathsf{pc}(t)$ is undefined, and

$$\sigma = \varsigma|_t\ \wedge\ \sigma_1 = \varsigma_1|_t\ \wedge\ \sigma_2 = \varsigma_2|_t.$$

Then $\sigma = \sigma_1 * \sigma_2$ and $\sigma \xrightarrow{\varphi}_{t,\mathcal{P}(\mathcal{L})} \sigma'$. We now make a case-split on which of rules (8)–(19) is used to derive φ.

- Rule (8) such that $\varphi = (t, c) \in \mathsf{PAct}$ is a client action in τ. Then, there exist $v, v' \in \mathsf{Node}$, $\alpha \in \mathsf{Pos}$, $l \in \mathsf{Foot}$, $\theta, \theta', \theta_1, \theta_2 \in \mathsf{State}$, such that (v, c, v') is in the control-flow relation of $\mathcal{P}$ and
$$\begin{aligned}\sigma_1 = (\alpha v, \theta_1, \delta(\theta_2) * l)\ \wedge\\ \sigma_2 = (v_{\mathsf{mgc}}^t, \theta_2, l)\ \wedge\\ \sigma = (\alpha v, \theta, l)\ \wedge\ \theta = \theta_1 * \theta_2;\end{aligned} \tag{22}$$
$$\sigma' = (\alpha v', \theta') \wedge \theta' \in f_c(\theta).$$
Then by (2) we have
$$\theta' \in f_c(\theta) = f_c(\theta_1 * \theta_2) = f_c(\theta_1) * \{\theta_2\}.$$
Thus, $\theta' = \theta_1' * \theta_2$, for some $\theta_1' \in f_c(\theta_1)$. Let
$$\sigma_1' = (\alpha v', \theta_1', \delta(\theta_2) * l),$$
$\sigma_2' = \sigma_2$, $\eta_{i+1} = \eta_i\varphi$ and $\xi_{i+1} = \xi_i$. Then $\sigma' = \sigma_1' * \sigma_2'$ and
$$\sigma_1 \xrightarrow{\varphi}_{t,\mathcal{P}} \sigma_1'.$$
Let $\varsigma_1' = \mathsf{combine}_t(\sigma_1', \mathsf{client}_{\mathsf{dom}(\Gamma_1)}(\mathsf{pc}), \mathsf{client}_{\mathsf{dom}(\Gamma_1),\alpha_0',t}(\kappa'))$ and $\varsigma_2' = \varsigma_2$, then
$$\varsigma_1 \xrightarrow{\varphi}_{\mathcal{P}} \varsigma_1'$$
and $\varsigma' = \varsigma_1' * \varsigma_2'$. We also have
$$\begin{aligned}&\mathsf{client}_{\mathsf{dom}(\Gamma_1)}(\tau_i\varphi) = \mathsf{client}_{\mathsf{dom}(\Gamma_1)}(\tau_i)\varphi =\\ &\quad\mathsf{ierase}_{(\mathsf{dom}(\Gamma_1)-\mathsf{dom}(\Gamma_0))\cap\mathsf{dom}(\Gamma_2)}(\eta_i)\varphi =\\ &\quad\mathsf{ierase}_{(\mathsf{dom}(\Gamma_1)-\mathsf{dom}(\Gamma_0))\cap\mathsf{dom}(\Gamma_2)}(\eta_i\varphi) =\\ &\qquad\mathsf{ierase}_{(\mathsf{dom}(\Gamma_1)-\mathsf{dom}(\Gamma_0))\cap\mathsf{dom}(\Gamma_2)}(\eta_{i+1})\end{aligned}$$
and
$$\begin{aligned}&\mathsf{erase}_{\mathsf{dom}(\Gamma_1)-\mathsf{dom}(\Gamma_0)}(\mathsf{lib}_{\mathsf{dom}(\Gamma_1)}(\tau_i\varphi)) =\\ &\quad\mathsf{erase}_{\mathsf{dom}(\Gamma_1)-\mathsf{dom}(\Gamma_0)}(\mathsf{lib}_{\mathsf{dom}(\Gamma_1)}(\tau_i)) =\\ &\qquad\mathsf{erase}_{\mathsf{dom}(\Gamma_1)-\mathsf{dom}(\Gamma_0)}(\eta_i) =\\ &\qquad\qquad\mathsf{erase}_{\mathsf{dom}(\Gamma_1)-\mathsf{dom}(\Gamma_0)}(\eta_{i+1}),\end{aligned}$$
from which the required follows.
- Rules (10) or (11) such that $\varphi \in \mathsf{CallRetAct}$ is a client action in τ. These cases are handled similarly to the previous one.
- Rules (8), (10) or (11) such that $\varphi \in \mathsf{PAct} \cup \mathsf{CallRetAct}$ is a library action in τ. These case are also handled similarly to the case of a client PAct. We establish that
$$\varsigma_2 \xrightarrow{\varphi}_{\mathcal{L}} \varsigma_2'$$
for some ς_2' such that $\varsigma' = \varsigma_1 * \varsigma_2'$ and let $\varsigma_1' = \varsigma_1$, $\eta_{i+1} = \eta_i$ and $\xi_{i+1} = \xi_i\varphi$. Since
$$\begin{aligned}&\mathsf{client}_{\mathsf{dom}(\Gamma_1)}(\tau_i\varphi) = \mathsf{client}_{\mathsf{dom}(\Gamma_1)}(\tau_i) =\\ &\quad\mathsf{ierase}_{(\mathsf{dom}(\Gamma_1)-\mathsf{dom}(\Gamma_0))\cap\mathsf{dom}(\Gamma_2)}(\eta_i) =\\ &\qquad\mathsf{ierase}_{(\mathsf{dom}(\Gamma_1)-\mathsf{dom}(\Gamma_0))\cap\mathsf{dom}(\Gamma_2)}(\eta_{i+1})\end{aligned}$$

[1] Here we use the equality that means both sides are defined and equal, or both are undefined.

and

$$\mathsf{erase}_{\mathsf{dom}(\Gamma_1)-\mathsf{dom}(\Gamma_0)}(\mathsf{lib}_{\mathsf{dom}(\Gamma_1)}(\tau_i\varphi)) = \mathsf{erase}_{\mathsf{dom}(\Gamma_1)-\mathsf{dom}(\Gamma_0)}(\mathsf{lib}_{\mathsf{dom}(\Gamma_1)}(\tau_i)\varphi) = \mathsf{erase}_{\mathsf{dom}(\Gamma_1)-\mathsf{dom}(\Gamma_0)}(\xi_i)\varphi = \mathsf{erase}_{\mathsf{dom}(\Gamma_1)-\mathsf{dom}(\Gamma_0)}(\xi_i\varphi) = \mathsf{erase}_{\mathsf{dom}(\Gamma_1)-\mathsf{dom}(\Gamma_0)}(\xi_{i+1}),$$

this implies our claim.

- Rule (10) such that $\varphi = (t, \mathsf{call}\ m)$ is an outermost call to $\mathcal{L}$ in τ and $m \in \mathsf{dom}(\Gamma_1) - \mathsf{dom}(\Gamma_0)$. Then there exist $v, v' \in \mathsf{Node}$, $\alpha \in \mathsf{Pos}$, $l \in \mathsf{Foot}$, $\theta, \theta_1, \theta_2 \in \mathsf{State}$, such that (v, m, v') is in the control-flow relation of $\mathcal{P}$, (B.3) is fulfilled and

$$\sigma' = (\alpha v' \mathsf{start}_m, \theta, l).$$

Since $\mathcal{P}$ is safe, we have $\theta_1 = \theta_1' * \theta_p$ for some $\theta_p \in p_t$, where $(\Gamma_1 - \Gamma_0)(m) = (p, \_)$. By Proposition 17, $\{\theta_1\} * \delta(\theta_2) * l \neq \emptyset$, hence, $l * \delta(\theta_2 * \theta_p) \neq \emptyset$. Let

$$\sigma_1' = (\alpha v_m, \theta_1', l * \delta(\theta_2 * \theta_p)) \wedge \sigma_2' = (v_{\mathsf{mgc}}\mathsf{start}_m, \theta_2 * \theta_p, l),$$

so that $\sigma_1' * \sigma_2' = \sigma'$ and

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

Let

$$\varsigma_1' = \mathsf{combine}_t(\sigma_1', \mathsf{client}_{\mathsf{dom}(\Gamma_1)}(\mathsf{pc}), \mathsf{client}_{\mathsf{dom}(\Gamma_1),\alpha_0',t}(\kappa')), \quad \varsigma_2' = \mathsf{combine}_t(\sigma_2', \mathsf{lib}_{\mathsf{dom}(\Gamma_1)}(\mathsf{pc}), \mathsf{lib}_{\mathsf{dom}(\Gamma_1),\alpha_0',t}(\kappa')),$$

then

$$\varsigma_1 \xrightarrow{\varphi}_{\mathcal{P}} \varsigma_1' \ \wedge\ \varsigma_2 \xrightarrow{\varphi}_{\mathcal{L}} \varsigma_2'$$

and $\varsigma' = \varsigma_1' * \varsigma_2'$. 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 $\mathsf{history}(\eta_{i+1}) = \mathsf{history}(\xi_{i+1})$,

$$\mathsf{client}_{\mathsf{dom}(\Gamma_1)}(\tau_i\varphi) = \mathsf{client}_{\mathsf{dom}(\Gamma_1)}(\tau_i)\varphi = \mathsf{ierase}_{(\mathsf{dom}(\Gamma_1)-\mathsf{dom}(\Gamma_0))\cap\mathsf{dom}(\Gamma_2)}(\eta_i)\varphi = \mathsf{ierase}_{(\mathsf{dom}(\Gamma_1)-\mathsf{dom}(\Gamma_0))\cap\mathsf{dom}(\Gamma_2)}(\eta_i(t, \mathsf{call}\ m(\theta_p))) = \mathsf{ierase}_{(\mathsf{dom}(\Gamma_1)-\mathsf{dom}(\Gamma_0))\cap\mathsf{dom}(\Gamma_2)}(\eta_{i+1})$$

and

$$\mathsf{erase}_{\mathsf{dom}(\Gamma_1)-\mathsf{dom}(\Gamma_0)}(\mathsf{lib}_{\mathsf{dom}(\Gamma_1)}(\tau_i\varphi)) = \mathsf{erase}_{\mathsf{dom}(\Gamma_1)-\mathsf{dom}(\Gamma_0)}(\mathsf{lib}_{\mathsf{dom}(\Gamma_1)}(\tau_i)\varphi) = \mathsf{erase}_{\mathsf{dom}(\Gamma_1)-\mathsf{dom}(\Gamma_0)}(\xi_i)\varphi = \mathsf{erase}_{\mathsf{dom}(\Gamma_1)-\mathsf{dom}(\Gamma_0)}(\xi_i(t, \mathsf{call}\ m(\theta_p))) = \mathsf{erase}_{\mathsf{dom}(\Gamma_1)-\mathsf{dom}(\Gamma_0)}(\xi_{i+1}).$$

Hence, the required follows.

- Rule (11) such that $\varphi = (t, \mathsf{ret}\ m)$ is a return from an outermost call to $\mathcal{L}$ in τ and $m \in \mathsf{dom}(\Gamma_1) - \mathsf{dom}(\Gamma_0)$. Then there exist $v' \in \mathsf{Node}$, $\alpha \in \mathsf{Pos}$, $l \in \mathsf{Foot}$, $\theta, \theta_1, \theta_2 \in \mathsf{State}$, such that

$$\sigma = (\alpha v' \mathsf{end}_m, \theta, l) \wedge \sigma' = (\alpha v', \theta, l) \ \wedge\ \theta = \theta_1 * \theta_2 \ \wedge \sigma_1 = (\alpha v' v_m, \theta_1, \delta(\theta_2) * l) \ \wedge \sigma_2 = (v_{\mathsf{mgc}}^t \mathsf{end}_m, \theta_2, l).$$

Since $\mathcal{L}$ is safe, we have $\theta_2 = \theta_2' * \theta_q$ for some $\theta_q \in q_t$, where $(\Gamma_1 - \Gamma_0)(m) = (\_, q)$. By Proposition 17, $\{\theta\} * l \neq \emptyset$, i.e., $\{\theta_1 * \theta_2' * \theta_q\} * l \neq \emptyset$. Then $\theta_1 * \theta_q$ is defined and so is $(\delta(\theta_2) * l) \setminus \delta(\theta_q) = \delta(\theta_2') * l$. Hence, for

$$\sigma_1' = (\alpha v', \theta_1 * \theta_q, (\delta(\theta_2) * l) \setminus \delta(\theta_q)) \wedge \sigma_2' = (v_{\mathsf{mgc}}, \theta_2', l),$$

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

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

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.

- Rule (15) such that $\varphi = (t, \mathsf{call}\ m(\theta_p))$ is a client action in τ, where $\theta_p \in p_t$ and $(\Gamma_2 - \Gamma_1)(m) = (p, \_)$. Then there exist $l \in \mathsf{Foot}$, $\theta, \theta_1, \theta_2 \in \mathsf{State}$, such that

$$\sigma = (w_{\mathsf{mgc}}^t, \theta, l) \ \wedge \sigma' = (w_{\mathsf{mgc}}^t \mathsf{start}_m, \theta * \theta_p, l) \ \wedge\ \theta = \theta_1 * \theta_2 \ \wedge \sigma_1 = (w_{\mathsf{mgc}}^t, \theta_1, \delta(\theta_2) * l) \ \wedge \sigma_2 = (v_{\mathsf{mgc}}^t, \theta_2, l). \tag{23}$$

and $\delta(\theta_1) * l * \delta(\theta_2) * \delta(\theta_p) = \delta(\theta) * l * \delta(\theta_p) \neq \emptyset$. Let

$$\sigma_1' = (v_{\mathsf{mgc}}^t \mathsf{start}_m, \theta_1 * \theta_p, \delta(\theta_2) * l)$$

and $\sigma_2' = \sigma_2$. Then $\sigma_1 \xrightarrow{\varphi}_{t,\mathcal{P}} \sigma_1'$ and $\sigma' = \sigma_1' * \sigma_2'$. We now show the required like in the case of a client PAct.

- Rule (15) such that $\varphi = (t, \mathsf{call}\ m(\theta_p))$ is a client action in τ, where $\theta_p \in p_t$ and $(\Gamma_2 \cap \Gamma_1 - \Gamma_0)(m) = (p, \_)$. Then there exist $l \in \mathsf{Foot}$, $\theta, \theta_1, \theta_2 \in \mathsf{State}$, such that (23) holds and $\delta(\theta_1) * l * \delta(\theta_2) * \delta(\theta_p) = \delta(\theta) * l * \delta(\theta_p) \neq \emptyset$. Let

$$\sigma_1' = (v_{\mathsf{mgc}}^t v_m, \theta_1, \delta(\theta_2) * l * \delta(\theta_p)) \quad \sigma_2' = (v_{\mathsf{mgc}}^t \mathsf{start}_m, \theta_2 * \theta_p, l).$$

Then $\sigma_1 \xrightarrow{\varphi}_{t,\mathcal{P}} \sigma_1'$, $\sigma_2 \xrightarrow{\varphi}_{t,\mathcal{P}} \sigma_2'$ and $\sigma' = \sigma_1' * \sigma_2'$.

- Rule (16) such that $\varphi = (t, \mathsf{ret}\ m(\theta_q))$, where $\theta_q \in q_t$ and $(\Gamma_2 - \Gamma_1)(m) = (\_, q)$. Then there exist $l \in \mathsf{Foot}$, $\theta, \theta', \theta_1, \theta_2 \in \mathsf{State}$, such that

$$\sigma = (w_{\mathsf{mgc}}^t \mathsf{end}_m, \theta, l) \ \wedge \sigma' = (w_{\mathsf{mgc}}^t, \theta', l) \ \wedge\ \theta = \theta_1 * \theta_2 = \theta' * \theta_q \ \wedge \sigma_1 = (w_{\mathsf{mgc}}^t \mathsf{end}_m, \theta_1, \delta(\theta_2) * l) \ \wedge \sigma_2 = (v_{\mathsf{mgc}}^t, \theta_2, l)$$

Since $\mathcal{P}$ is safe, we have $\theta_1 = \theta_1' * \theta_q'$ for some $\theta_q' \in q_t$. Then $\theta_1' * \theta_q' * \theta_2 = \theta' * \theta_q$. Since q is precise and $*$ is cancellative, this entails $\theta_q' = \theta_q$ and $\theta_1' * \theta_2 = \theta'$. Let

$$\sigma_1' = (w_{\mathsf{mgc}}^t, \theta_1', l)$$

and $\sigma_2' = \sigma_2$, then $\sigma_1 \xrightarrow{\varphi}_{t,\mathcal{P}} \sigma_1'$ and $\sigma' = \sigma_1' * \sigma_2'$.

- Rule (16) such that $\varphi = (t, \mathsf{ret}\ m(\theta_q))$, where $\theta_q \in q_t$ and $(\Gamma_2 \cap \Gamma_1 - \Gamma_0)(m) = (\_, q)$. Then there exist $l \in \mathsf{Foot}$, $\theta, \theta', \theta_1, \theta_2 \in \mathsf{State}$, such that

$$\sigma = (w_{\mathsf{mgc}}^t \mathsf{end}_m, \theta, l) \ \wedge \sigma' = (w_{\mathsf{mgc}}^t, \theta', l) \ \wedge\ \theta = \theta_1 * \theta_2 = \theta' * \theta_q \ \wedge \sigma_1 = (w_{\mathsf{mgc}}^t v_m, \theta_1, \delta(\theta_2) * l) \ \wedge \sigma_2 = (v_{\mathsf{mgc}}^t \mathsf{end}_m, \theta_2, l)$$

Since $\mathcal{L}$ is safe, we have $\theta_2 = \theta_2' * \theta_q'$ for some $\theta_q' \in q_t$. Then $\theta_1 * \theta_q' * \theta_2' = \theta' * \theta_q$. Since q is precise and $*$ is cancellative, this entails $\theta_q' = \theta_q$ and $\theta_1 * \theta_2' = \theta'$. Let

$$\sigma_1' = (w_{\mathsf{mgc}}^t, \theta_1, \delta(\theta_2') * l) \quad \sigma_2' = (v_{\mathsf{mgc}}^t, \theta_2', l),$$

then $\sigma_1 \xrightarrow{\varphi}_{t,\mathcal{P}} \sigma_1'$ and $\sigma' = \sigma_1' * \sigma_2'$.

- Rule (12) such that $\varphi = (t, \mathsf{call}\ m(\theta_p))$ is a library action in τ, where $\Gamma_0(m) = (p, \_)$ and $\theta_p \in p_t$. Then, there exist $v, v', v_{m'} \in \mathsf{Node}$, $\alpha, \beta \in \mathsf{Pos}$, $l \in \mathsf{Foot}$, $\theta, \theta', \theta_1, \theta_2 \in \mathsf{State}$, such that (v, m, v') is in the control-flow relation of $\mathcal{L}$ and

$$\sigma = (\alpha\beta v, \theta, l) \ \wedge \sigma' = (\alpha\beta v' v_m, \theta', l * \delta(\theta_p)).\theta = \theta_1 * \theta_2 = \theta' * \theta_p \ \wedge \sigma_1 = (\alpha v_{m'}, \theta_1, \delta(\theta_2) * l) \ \wedge \sigma_2 = (v_{\mathsf{mgc}}^t \beta v, \theta_2, l).$$

Since $\mathcal{L}$ is safe, we have $\theta_2 = \theta_2' * \theta_p'$, where $\theta_p' \in p_t$. Besides, p is precise and $*$ is cancellative, so we have $\theta_p' = \theta_p$ and $\theta_1 * \theta_2' = \theta'$. Hence, for

$$\sigma_2' = (v_{\mathsf{mgc}}^t \beta v' v_m, \theta_2', l * \delta(\theta_p))$$

and $\sigma_1' = \sigma_1$, we have $\sigma_1' * \sigma_2' = \sigma'$ and $\sigma_2 \xrightarrow{\varphi}_{t,\mathcal{L}} \sigma_2'$.

- Rule (14) such that $\varphi = (t, \mathsf{ret}\ m(\theta_q))$ is a library action in τ, where $\Gamma_0(m) = (\_, q)$ and $\theta_q \in q_t$. Then, there exist $v', v_{m'} \in \mathsf{Node}$, $\alpha, \beta \in \mathsf{Pos}$, $l \in \mathsf{Foot}$, $\theta, \theta_1, \theta_2 \in \mathsf{State}$, such

that

$$\sigma = (\alpha\beta v' v_m, \theta, l) \ \wedge$$
$$\sigma' = (\alpha\beta v', \theta * \theta_q, l \setminus \delta(\theta_q)) \ \wedge \ \theta = \theta_1 * \theta_2 \ \wedge$$
$$\sigma_1 = (\alpha v_{m'}, \theta_1, \delta(\theta_2) * l) \ \wedge$$
$$\sigma_2 = (v^t_{\mathsf{mgc}} \beta v' v_m, \theta_2, l).$$

Hence, for

$$\sigma_2' = (v^t_{\mathsf{mgc}} \beta v', \theta_2 * \theta_q, l \setminus \delta(\theta_q))$$

and $\sigma_1' = \sigma_1$, we have $\sigma_1' * \sigma_2' = \sigma'$ and $\sigma_2 \xrightarrow{\varphi}_{t,\mathcal{L}} \sigma_2'$.

- Rules (12) and (14) such that φ is a client action. These cases are handled similarly to the previous two.
- Rule (18) such that $\varphi = (t, \mathsf{call}\ m(\theta_p))$, where $(\Gamma_2 \cap \Gamma_0)(m) = (p, \_)$ and $\theta_p \in p_t$. Then, there exist $l \in \mathsf{Foot}$, $\theta, \theta_1, \theta_2 \in \mathsf{State}$, such that

$$\sigma = (w^t_{\mathsf{mgc}}, \theta, l) \ \wedge$$
$$\sigma' = (w^t_{\mathsf{mgc}} v_m, \theta, l * \delta(\theta_p)) \ \wedge \ \theta = \theta_1 * \theta_2 \ \wedge$$
$$\sigma_1 = (w^t_{\mathsf{mgc}}, \theta_1, \delta(\theta_2) * l) \ \wedge$$
$$\sigma_2 = (v^t_{\mathsf{mgc}}, \theta_2, l)$$

and $\delta(\theta) * l * \delta(\theta_p) \neq \emptyset$. Hence, for

$$\sigma_1' = (w^t_{\mathsf{mgc}} v_m, \theta_1, \delta(\theta_2) * l * \delta(\theta_p))$$
$$\sigma_2' = (v^t_{\mathsf{mgc}} v_m], \theta_2, l * \delta(\theta_p)),$$

we have $\sigma_1' * \sigma_2' = \sigma'$, $\sigma_1 \xrightarrow{\varphi}_{t,\mathcal{L}} \sigma_1'$ and $\sigma_2 \xrightarrow{\varphi}_{t,\mathcal{L}} \sigma_2'$.

- Rule (19) such that $\varphi = (t, \mathsf{ret}\ m(\theta_q))$, where $(\Gamma_2 \cap \Gamma_0)(m) = (\_, q)$ and $\theta_q \in q_t$. Then, there exist $l \in \mathsf{Foot}$, $\theta, \theta_1, \theta_2 \in \mathsf{State}$, such that

$$\sigma = (w^t_{\mathsf{mgc}} v_m, \theta, l) \ \wedge$$
$$\sigma' = (w^t_{\mathsf{mgc}}, \theta, l \setminus \delta(\theta_q)) \ \wedge \ \theta = \theta_1 * \theta_2 \ \wedge$$
$$\sigma_1 = (w^t_{\mathsf{mgc}} v_m, \theta_1, \delta(\theta_2) * l) \ \wedge$$
$$\sigma_2 = (v^t_{\mathsf{mgc}} v_m, \theta_2, l).$$

Hence, for

$$\sigma_1' = (w^t_{\mathsf{mgc}}, \theta_1, (\delta(\theta_2) * l) \setminus \delta(\theta_q))$$
$$\sigma_2' = (v^t_{\mathsf{mgc}}, \theta_2, l \setminus \delta(\theta_q)),$$

we have $\sigma_1' * \sigma_2' = \sigma'$, $\sigma_1 \xrightarrow{\varphi}_{t,\mathcal{L}} \sigma_1'$ and $\sigma_2 \xrightarrow{\varphi}_{t,\mathcal{L}} \sigma_2'$.

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

B.4 Proof of Lemma 13

In this section, we use the following notation. We denote derivations of traces in the semantics of Section 3 with $\mathcal{D}, \mathcal{E}, \ldots$. For two derivations $\mathcal{D}_1$ and $\mathcal{D}_2$ their concatenation $\mathcal{D}_1\mathcal{D}_2$ is defined when the last state in $\mathcal{D}_1$ is the same as the first state in $\mathcal{D}_2$. The concatenation glues the derivations at this state. Below we sometimes write $\sqsubseteq_\pi$ and $\lhd_M^\pi$ instead of $\sqsubseteq$ and $\lhd_M$ to make the bijection π used to establish the relations between histories or traces explicit.

Let $\mathsf{dom}(\Gamma) = M$. Take $\theta_0 \in \mathsf{State}$ and $l_1, l_2 \in \mathsf{Foot}$ such that $l_1 \succeq l_2$. Consider a trace τ such that $(\theta_0, l_1, \tau) \in [\![\mathcal{P}]\!]$ and the corresponding derivation $\mathcal{D}$ in the semantics of $\mathcal{P}$. Assume well-formed histories $H, H' \in \mathsf{History}$ of methods in M such that $\mathsf{history}_M(\tau) = H$, $H \sqsubseteq H'$ and H' is well-balanced from l_2 (H is well-balanced from l_1 by Proposition 7). We first prove that there exists a trace τ' such that $(\theta_1, l_1, \tau_2) \in [\![\mathcal{P}]\!]$ with a corresponding derivation $\mathcal{D}'$ in the the semantics of $\mathcal{P}$ such that $\mathsf{history}_M(\tau') = H'$ and $\tau \lhd_M \tau'$. To this end, we define a (possibly infinite) sequence of steps that transforms τ into τ' and $\mathcal{D}$ into $\mathcal{D}'$. In more detail, τ' and $\mathcal{D}'$ are constructed using a sequence of traces ξ_k such that $(\theta_0, l_1, \xi_k) \in [\![\mathcal{P}]\!]$ and their corresponding derivations $\mathcal{D}_k$, defined for every finite prefix H_k' of H' of length k as described below. Every trace ξ_k is such that for some prefix η_k of ξ_k we have $\mathsf{history}_M(\eta_k) = H_k'$ and $\mathsf{history}_M(\xi_k) \sqsubseteq_\pi H'$, where π is an identity on actions from H_k' in $\mathsf{history}_M(\xi_k)$. Let $\mathcal{D}_k'$ be the prefix of $\mathcal{D}_k$ corresponding to the derivation of η_k. Then, additionally, for all i, j with $i < j$, η_i is a prefix of η_j and $\mathcal{D}_i'$ is a prefix of $\mathcal{D}_j'$. Hence, the sequences of traces η_k and derivations $\mathcal{D}_k'$ have limits τ' and $\mathcal{D}'$ such that for every k, η_k is a prefix of τ', $\mathcal{D}_k'$ is a prefix of $\mathcal{D}'$ and $\mathsf{history}_M(\tau') = H'$. Then, as we show, $(\theta_0, l_1, \tau') \in [\![\mathcal{P}]\!]$, its derivation is $\mathcal{D}'$ and $\mathsf{history}_M(\tau') = H'$, $\tau \lhd_M \tau'$.

The transformations we perform on the trace τ might violate atomicity constraints temporarily. Therefore, actually, we construct the derivations $\mathcal{D}_i$ and $\mathcal{D}'$ and define $[\![\mathcal{P}]\!]$ above in the semantics given by the relation $\hookrightarrow_{\Gamma,\mathcal{P},\Gamma'}$: $\mathsf{NConfig} \times \mathsf{Act} \times \mathsf{NConfig}$, for

$$\mathsf{NConfig} = \{\top\} \cup (\mathsf{ThreadID} \rightharpoonup_{fin} \mathsf{Pos}) \times \mathsf{State} \times \mathsf{Foot},$$

which ignores the scheduling policy. The relation $\hookrightarrow_{\mathcal{P}}$ is defined like the one in Figure 2, but with rules (4)–(7) replaced by the following ones:

$$\frac{\alpha, \theta, l \xrightarrow{\varphi}_t \alpha', \theta', l'}{\mathsf{pc}[t : \alpha], \theta, l \overset{\varphi}{\hookrightarrow}_{\mathcal{P}} \mathsf{pc}[t : \alpha'], \theta', l}$$

$$\frac{\alpha, \theta, l \xrightarrow{\varphi}_t \top}{\mathsf{pc}[t : \alpha], \theta, l \overset{\varphi}{\hookrightarrow}_{\mathcal{P}} \top}$$

However, that the final trace τ' satisfies the atomicity constraints in the following sense. We say that τ' *respects the atomicity of* $\Gamma \vdash \mathcal{P} : \Gamma'$ if it does not interleave actions by any thread inside methods declared atomic in Γ or $\mathcal{P}$ with actions by other threads. It is easy to check that the transformations below indeed result in a trace τ' respecting the atomicity of $\Gamma \vdash \mathcal{P} : \Gamma'$ and H'. As the following proposition shows, in this case τ' is in fact derivable using $\longrightarrow$.

Proposition 18. *If*

$$(\mathsf{pc}_0, \theta_0, l_0) \overset{\tau'}{\hookrightarrow}_{\mathcal{P}} \sigma$$

for some $\sigma \neq \top$, and τ' respects the atomicity of $\Gamma \vdash \mathcal{P} : \Gamma'$, then

$$(\mathsf{pc}_0, \theta_0, l_0, \mathsf{ThreadID}) \overset{\tau'}{\longrightarrow}_{\mathcal{P}} \varsigma$$

for some $\varsigma \neq \top$.

The fact that $(\theta_0, l_2, \tau') \in [\![\mathcal{P}]\!]$, as required by the statement of the lemma, follows from the following proposition, proved at the end of this section.

Proposition 19 (Footprint reduction). *Assume $[\![\mathcal{P}]\!]$ is safe at (θ_0, l_1), $(\theta_0, l_1, \tau) \in [\![\mathcal{P}]\!]$, $l_2 \preceq l_1$ and $\mathsf{history}_M(\tau)$ be well-balanced from l_2. Then $(\theta_0, l_2, \tau) \in [\![\mathcal{P}]\!]$.*

To construct the sequence of traces, we let $\xi_0 = \tau$ and let the prefix η_0 contain all the client actions preceding the first call or return action in ξ_0. The trace ξ_{k+1} is constructed from the trace ξ_k by applying the following lemma for $\tau_1 = \eta_k$, $\tau_1\tau_2 = \xi_k$, $\mathcal{E}_1 = \mathcal{D}_k'$, $\mathcal{E}_1\mathcal{E}_2 = \mathcal{D}_k$, $H_1 = H_k'$ and $H_1\psi H_2 = H'$.

Lemma 20. *Consider a well-formed history $H_1\psi H_2$ of methods in M, where $\psi \in \mathsf{ECallRetAct}_M$, and $(\theta_0, l_1, \tau_1\tau_2) \in [\![\mathcal{P}]\!]$ with a derivation $\mathcal{E}_1\mathcal{E}_2$ (where $\mathcal{E}_1$ derives τ_1 and $\mathcal{E}_2$ derives τ_2) such that*

$$\mathsf{history}_M(\tau_1) = H_1, \tag{24}$$
$$\mathsf{history}_M(\tau_1\tau_2) \sqsubseteq_\pi H_1\psi H_2, \tag{25}$$

where π is an identity on actions from H_1 in $\mathsf{history}_M(\tau_1\tau_2)$. Then there exist traces τ_2' and τ_2'' and their derivations $\mathcal{E}_2'$ and $\mathcal{E}_2''$ such that $(\theta_0, l_1, \tau_1\tau_2'\tau_2'') \in [\![\mathcal{P}]\!]$, the corresponding derivation is $\mathcal{E}_1\mathcal{E}_2'\mathcal{E}_2''$ and

$$\tau_1\tau_2 \lhd_M^\rho \tau_1\tau_2'\tau_2'', \tag{26}$$
$$\mathsf{history}_M(\tau_1\tau_2') = H_1\psi, \tag{27}$$
$$\mathsf{history}_M(\tau_1\tau_2'\tau_2'') \sqsubseteq_{\pi'} H_1\psi H_2, \tag{28}$$

where π' is an identity on actions from $H_1\psi$ in $\mathsf{history}_M(\tau_1\tau_2'\tau_2'')$ and ρ is an identity on actions from τ_1 in $\tau_1\tau_2$.

To prove Lemma 20, we transform $\tau_1\tau_2$ into $\tau_1\tau_2'\tau_2''$, respectively, $\mathcal{D}_1\mathcal{D}_2$ into $\mathcal{D}_1\mathcal{D}_2'$ by applying a finite number of transformations that preserve their properties of interest, described in Proposition 21, respectively, Proposition 22 below.

Proposition 21. *Let τ be a trace and H a well-formed history such that $\mathsf{history}_M(\tau) \sqsubseteq_\pi H$. Then swapping any two adjacent actions*

φ_1 and φ_2 in τ executed by different threads such that
1. $\varphi_1 \in \mathsf{ECallAct}_M$ and $\varphi_2 \in \mathsf{Act} - \mathsf{ERetAct}_M$; or
2. $\varphi_2 \in \mathsf{ERetAct}_M$ and if $\varphi_1 \in \mathsf{ECallAct}_M$, then φ_2 precedes φ_1 in H,
yields a trace τ' such that $\tau \lhd^{\rho}_M \tau'$ and $\mathsf{history}_M(\tau') \sqsubseteq_{\pi'} H$.

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

The bijection ρ is defined as follows. Let i be the index of φ_1 in τ. Then $\rho'(i+1) = \rho(i)$, $\rho'(i) = \rho(i+1)$ and $\rho'(k) = \rho(k)$ for $k \notin \{i, i+1\}$.

Proposition 22. *Consider actions $\varphi_1, \varphi_2 \in \mathsf{Act}$ by different threads such that any of the following is true:*
- *$\varphi_1 \in \mathsf{ECallAct}_M$, $\varphi_2 \in \mathsf{Act} - \mathsf{ERetAct}_M$;*
- *$\varphi_1 \in \mathsf{Act} - \mathsf{ECallAct}_M$, $\varphi_2 \in \mathsf{ERetAct}_M$.*

If for $\sigma_0, \sigma_1, \sigma_2 \in \mathsf{Config} - \{\top\}$ we have

$$\sigma_0 \overset{\varphi_1}{\hookrightarrow}_{\mathcal{P}} \sigma_1 \overset{\varphi_2}{\hookrightarrow}_{\mathcal{P}} \sigma_2, \tag{29}$$

then for some $\sigma_1' \in \mathsf{NConfig} - \{\top\}$

$$\sigma_0 \overset{\varphi_2}{\hookrightarrow}_{\mathcal{P}} \sigma_1' \overset{\varphi_1}{\hookrightarrow}_{\mathcal{P}} \sigma_2.$$

Proof. The proof proceeds by case analysis on rules of operational semantics that can produce φ_1 and φ_2. We consider only a couple of illustrative cases.

- Rules (12) and (8) such that that $\varphi_1 \in \mathsf{ECallAct}_M$ and $\varphi_2 \in \mathsf{PAct}$. Then there exist $t_1, t_2 \in \mathsf{ThreadID}$, $v_1, v_1', v_2, v_2' \in \mathsf{Node}$, $\mathsf{pc} \in \mathsf{ThreadID} \rightharpoonup \mathsf{Pos}$, $\alpha_1, \alpha_2 \in \mathsf{Pos}$, $l \in \mathsf{Foot}$, $\theta_1, \theta_2, \theta_p \in \mathsf{State}$, such that (v_1, m, v_1') and (v_2, c, v_2') are in the control-flow relation of $\mathcal{P}$, $\Gamma(m) = (p, \_)$ and

$$\begin{gathered}\sigma_0 = (\mathsf{pc}[t_1 : \alpha_1 v_1][t_2 : \alpha_2 v_2], \theta_1 * \theta_p, l) \ \wedge \\ \sigma_1 = (\mathsf{pc}[t_1 : \alpha_1 v_1][t_2 : \alpha_2 v_2' v_m], \theta_1, l * \delta(\theta_p)) \ \wedge \\ \sigma_2 = (\mathsf{pc}[t_1 : \alpha_1 v_1'][t_2 : \alpha_2 v_2' v_m], \theta_2, l * \delta(\theta_p)) \ \wedge \\ \theta_p \in p_{t_1} \ \wedge \ \theta_2 \in f_c(\theta_1).\end{gathered}$$

  Since $\theta_1 * \theta_p$ is defined, by (3) so is $\theta_2 * \theta_p$. Then from (2) we get

$$\theta_2 * \theta_p \in f_c(\theta_1) * \theta_p = f_c(\theta_1 * \theta_p).$$

  Then the required holds for

$$\sigma_1' = (\mathsf{pc}[t_1 : \alpha_1 v_1'][t_2 : \alpha_2 v_2'], \theta_2 * \theta_p, l).$$

- Rules (12) and (15) such that $\varphi_1, \varphi_2 \in \mathsf{ECallAct}$. Then there exist $t_1, t_2 \in \mathsf{ThreadID}$, $v_1, v_1' \in \mathsf{Node}$, $\mathsf{pc} \in \mathsf{ThreadID} \rightharpoonup \mathsf{Pos}$, $\alpha_1, \alpha_2 \in \mathsf{Pos}$, $l \in \mathsf{Foot}$, $\theta_1, \theta_2, \theta_p^1, \theta_p^2 \in \mathsf{State}$, such that (v_1, m_1, v_1') is in the control-flow relation of $\mathcal{P}$, $\Gamma(m_1) = (p^1, \_)$, $\Gamma'(m_2) = (p^2, \_)$ and

$$\begin{gathered}\sigma_0 = (\mathsf{pc}[t_1 : \alpha_1 v_1][t_2 : w^t_{\mathsf{mgc}}], \theta_1 * \theta_p^1, l) \ \wedge \\ \sigma_1 = (\mathsf{pc}[t_1 : \alpha_1 v_1' v_{m_1}][t_2 : w^t_{\mathsf{mgc}}], \theta_1, l * \delta(\theta_p^1)) \ \wedge \\ \sigma_2 = (\mathsf{pc}[t_1 : \alpha_1 v_1' v_{m_1}][t_2 : w^t_{\mathsf{mgc}} \mathsf{start}_{m_2}], \theta_1 * \theta_p^2, l * \delta(\theta_p^1)) \wedge \\ \theta_p^1 \in p^1_{t_1} \ \wedge \ \theta_p^2 \in p^2_{t_2} \ \wedge \ \{\theta_1\} * \{\theta_p^1\} * l * \delta(\theta_p^2) \neq \emptyset.\end{gathered}$$

  Then the required holds for

$$\sigma_1' = (\mathsf{pc}[t_1 : \alpha_1 v_1][t_2 : w^t_{\mathsf{mgc}} \mathsf{start}_{m_2}], \theta_1 * \theta_p^1 * \theta_p^2, l).$$

□

Note that the above proposition does not allow swapping a call followed by a return. This case is very subtle and relies crucially on the well-balancedness property of histories under consideration. The following lemma proves a key property showing that the swapping can be done under certain circumstances, which are in fact fulfilled in the context where we apply it in the proof of Lemma 20.

Lemma 23. *Consider histories*

$$H_1 = HH_1'(t_1, \mathsf{call}\ m_1(\theta_1))(t_2, \mathsf{ret}\ m_2(\theta_2))H_1'',$$
$$H_2 = H(t_2, \mathsf{ret}\ m_2(\theta_2))H_2'(t_1, \mathsf{call}\ m_1(\theta_1))H_2''$$

of methods in M, well-balanced from l_1 and l_2, respectively and assume that $l_2 \preceq l_1$, $t_1 \neq t_2$ and $H_1 \sqsubseteq_\pi H_2$, where π is an identity on H and maps the $(t_1, \mathsf{call}\ m_1(\theta_1))$ and $(t_2, \mathsf{ret}\ m_2(\theta_2))$ actions in H_1 to the corresponding actions in H_2. Then the history

$$\tilde{H}_1 = HH_1'(t_2, \mathsf{ret}\ m_2(\theta_2))(t_1, \mathsf{call}\ m_1(\theta_1))H_1''$$

is also well-balanced from l_1.

Proof. Every return action in H_1' precedes $(t_1, \mathsf{call}\ m_1(\theta_1))$ in H_1. Since $H_1 \sqsubseteq H_2$, it follows that every such action is in H_2'. For the same reason, every call action in H_1'' that is also in H_2' follows all such return actions in H_2. Thus, there is a prefix H_2''' of H_2' whose actions are a subset of the actions in H_1', including all the returns and some of the calls from H_1'.

Since H_2 is well-balanced from l_2,

$$\mathsf{run}(H(t_2, \mathsf{ret}\ m_2(\theta_2))H_2''', l_2) = \mathsf{run}(H_2''', \mathsf{run}(H, l_2) \setminus \delta(\theta_2))$$

is defined (see Definition 6). Since H_1 is well-balanced from l_1,

$$\mathsf{run}(HH_1', l_1) = \mathsf{run}(H_1', \mathsf{run}(H, l_1))$$

is defined.

We have assumed that $l_2 \preceq l_1$, which implies $\mathsf{run}(H, l_2) \preceq \mathsf{run}(H, l_1)$. From the above it follows that $\mathsf{run}(H, l_2) \setminus \delta(\theta_2)$ is defined, hence, $\mathsf{run}(H, l_1) \setminus \delta(\theta_2)$ is also defined and $\mathsf{run}(H, l_2) \setminus \delta(\theta_2) \preceq \mathsf{run}(H, l_1) \setminus \delta(\theta_2)$. From this, the above-stated relationship between actions in H_1' and H_2', and the associativity property of $*$ and $\setminus$ formulated in Appendix A, we obtain that $\mathsf{run}(H_1', \mathsf{run}(H, l_1) \setminus \delta(\theta_2))$ is defined. But this means that the history $HH_1'(t_2, \mathsf{ret}\ m_2(\theta_2))$ is well-balanced from l_1.

Since H_1 is well-balanced from l_1,

$$\mathsf{run}(HH_1', l_1) * \delta(\theta_1)$$

is defined. Then so is

$$(\mathsf{run}(HH_1', l_1) \setminus \delta(\theta_2)) * \delta(\theta_1).$$

Hence, $HH_1'(t_2, \mathsf{ret}\ m_2(\theta_2))(t_1, \mathsf{call}\ m_1(\theta_1))$ is well-balanced from l_1. Besides

$$\begin{aligned}\mathsf{run}(HH_1'(t_2, \mathsf{ret}\ m_2(\theta_2))(t_1, \mathsf{call}\ m_1(\theta_1)), l_1) = \\ \mathsf{run}(HH_1'(t_1, \mathsf{call}\ m_1(\theta_1))(t_2, \mathsf{ret}\ m_2(\theta_2)), l_1),\end{aligned}$$

so the history $\tilde{H}_1$ is also well-balanced from l_1. □

The following corollary of the lemma justifies the transformation.

Corollary 24. *Consider a trace $(\theta_0, l_1, \tau_1) \in [\![\mathcal{P}]\!]$ and a history H_2 of methods in M such that $\mathsf{history}_M(\tau_1) = H_1$ and H_1 and H_2 satisfy the conditions of Lemma 23. Let*

$$\sigma_0 \overset{\tau\tau_1'}{\hookrightarrow}{}^*_{\mathcal{P}}\, \sigma_1 \xrightarrow{(t_1, \mathsf{call}\ m_1(\theta_1))}{}^*_{\mathcal{P}}\, \sigma_2 \xrightarrow{(t_2, \mathsf{ret}\ m_2(\theta_2))}{}^*_{\mathcal{P}}\, \sigma_3 \overset{\tau_1''}{\hookrightarrow}{}^*_{\mathcal{P}}\, \sigma_4$$

be the derivation of τ_1, where $t_1 \neq t_2$. Then for some σ_2' the following is also a valid derivation:

$$\sigma_0 \overset{\tau\tau_1'}{\hookrightarrow}{}^*_{\mathcal{P}}\, \sigma_1 \xrightarrow{(t_2, \mathsf{ret}\ m_2(\theta_2))}{}^*_{\mathcal{P}}\, \sigma_2' \xrightarrow{(t_1, \mathsf{call}\ m_1(\theta_1))}{}^*_{\mathcal{P}}\, \sigma_3 \overset{\tau_1''}{\hookrightarrow}{}^*_{\mathcal{P}}\, \sigma_4.$$

Proof. We can assume that for some $v_1, v_1' \in \mathsf{Node}$, $\mathsf{pc} \in \mathsf{ThreadID} \rightharpoonup \mathsf{Pos}$, $\alpha_1, \alpha_2 \in \mathsf{Pos}$, $l \in \mathsf{Foot}$, $\theta \in \mathsf{State}$, (v_1, m_1, v_1') is in the control-flow relation of $\mathcal{P}$ and

$$\begin{gathered}\sigma_1 = (\mathsf{pc}[t_1 : \alpha_1 v_1][t_2 : \alpha_2 v_{m_2}], \theta * \theta_1, l) \wedge \\ \sigma_2 = (\mathsf{pc}[t_1 : \alpha_1 v_1' v_{m_1}][t_2 : \alpha_2 v_{m_2}], \theta, l * \delta(\theta_1)) \wedge \\ \sigma_3 = (\mathsf{pc}[t_1 : \alpha_1 v_1' v_{m_1}][t_2 : \alpha_2], \theta * \theta_2, (l * \delta(\theta_1)) \setminus \delta(\theta_2)).\end{gathered}$$

By Lemma 23, $\mathsf{history}_M(\tau\tau_1'(t_2, \mathsf{ret}\ m_2(\theta_2))(t_1, \mathsf{call}\ m_1(\theta_1))\tau_1'')$ is well-balanced from the initial footprint in σ_0. Hence, $l \setminus \delta(\theta_2)$ and $(l \setminus \delta(\theta_2)) * \delta(\theta_1)$ are defined. Then by Proposition 16(3), $(l * \delta(\theta_1)) \setminus \delta(\theta_2) = (l \setminus \delta(\theta_2)) * \delta(\theta_1)$. By Proposition 17, $l * \{\theta * \theta_1\} \neq \emptyset$. Since $l \setminus \delta(\theta_2)$ is defined, so is $\theta * \theta_1 * \theta_2$. The above implies that

$$\sigma_2' = (\mathsf{pc}[t_1 : \alpha_1 v_1][t_2 : \alpha_2], \theta * \theta_1 * \theta_2, l \setminus \delta(\theta_2))$$

is defined and

$$\sigma_1 \xrightarrow{(t_2, \mathsf{ret}\ m_2(\theta_2))}{}^*_{\mathcal{P}}\, \sigma_2' \xrightarrow{(t_1, \mathsf{call}\ m_1(\theta_1))}{}^*_{\mathcal{P}}\, \sigma_3.$$

□

Proof of Lemma 20. From (24) and (25) it follows that $\tau_2 = \tau_3'\psi\tau_4'$ for some traces τ_3' and τ_4'. We have two cases.

1. $\psi \in \mathsf{ECallAct}_M$. Then τ_3' cannot contain an action $\varphi \in \mathsf{ERetAct}_M$, because in this case φ would precede ψ in $\mathsf{history}_M(\tau_1\tau_2)$. However, $\varphi \in H_2$ and, thus, ψ precedess φ in $H_1\psi H_2$, so this would contradict (25). Hence, there are no return actions in τ_3'. Moreover, since $\tau_1\tau_2$ is well-formed, for any action $\varphi = (t, \mathsf{call}\ m(\theta))$ in τ_3' there are no actions by the thread t in τ_3' following φ. Thus, we can move all actions from $\mathsf{ECallAct}$ in the subtrace τ_3' of $\tau_1\tau_2$ to the position right after ψ by swapping adjacent actions in $\tau_1\tau_2$ a finite number of times as described in Proposition 21(1) and transforming its derivation as described in Proposition 22 and Lemma 24. We thus obtain the trace $\tau_1\tau_3''\psi\tau_4''\tau_4'$, where τ_4'' consists of $\mathsf{ECallAct}_M$ actions in τ_3' and τ_3'' of the rest of actions in the subtrace. Conditions (26)–(28) then follow from Proposition 21(1) and the transitivity of $\lhd_M$ for $\tau_2' = \tau_3''\psi$ and $\tau_2'' = \tau_4''\tau_4'$.

2. $\psi \in \mathsf{ERetAct}_M$. Since the history $\mathsf{history}_M(\tau_1\tau_2)$ is well-formed, the matching call of ψ is in H_1. From (24) and (25) it then follows that this call is not in τ_3'. Furthermore, since the trace $\tau_1\tau_2$ is well-formed, the thread that executed ψ does not execute any actions in the subtrace in τ_3'. Thus, we can move the action ψ to the beginning of τ_3' by swapping adjacent actions in $\tau_1\tau_2$ a finite number of times as described in Proposition 21(2) and transforming its derivation as described in Proposition 22 and Lemma 24. We thus obtain the trace $\tau_1\psi\tau_3'\tau_4'$. Conditions (26)–(28) then follow from Proposition 21(2) and the transitivity of $\lhd_M$ for $\tau_2' = \psi$ $\tau_2'' = \tau_3'\tau_4'$. □

Now all is left is to prove Proposition 19.

Proof of Proposition 19. Consider $(\theta_0, l_1, \tau) \in [\![\mathcal{P}]\!]$. Then there exists a non-faulting derivation

$$(\mathsf{pc}_0, \theta_0, l_1, \mathsf{ThreadID}) \xrightarrow{\tau}{}^*_{\mathcal{P}} (\mathsf{pc}, \theta, l_1', \kappa)$$

of τ in the semantics of $\mathcal{P}$. Take $l_2 \preceq l_1$ and assume that $\mathsf{history}_M(\tau)$ is well-balanced from l_2. We prove by induction on the length of the derivation of τ that there exists a derivation

$$(\mathsf{pc}_0, \theta_0, l_2, \mathsf{ThreadID}) \xrightarrow{\tau}{}^*_{\mathcal{P}} (\mathsf{pc}, \theta, l_2', \kappa)$$

such that $\mathsf{run}(\mathsf{history}_M(\tau), l_2) = l_2'$ and $l_2' \preceq l_1'$.

The base case trivially follows from the definition of run. For the induction step, assume the above and

$$(\mathsf{pc}, \theta, l_1', \kappa) \xrightarrow{\varphi}_{\mathcal{P}} (\mathsf{pc}', \theta', l_1'', \kappa').$$

We need to show that

$$(\mathsf{pc}, \theta, l_2', \kappa) \xrightarrow{\varphi}_{\mathcal{P}} (\mathsf{pc}', \theta', l_2'', \kappa')$$

and $\mathsf{run}(\mathsf{history}_M(\tau\varphi), l_2) = l_2''$ and $l_2'' \preceq l_1''$.

We consider every rule in the operational semantics that can produce φ.

- Rule (12) such that $\varphi = (t, \mathsf{call}\ m(\theta_p))$, where $\Gamma(m) = (p, \_)$ and $\theta_p \in p_t$. Then $l_1'' = l_1' * \delta(\theta_p)$ and $\theta = \theta' * \theta_p$. Since $l_2' \preceq l_1'$, $l_2' * \delta(\theta_p) \neq \emptyset$ and $l_2' * \delta(\theta_p) \preceq l_1''$. Besides, $\mathsf{run}(\mathsf{history}_M(\tau_1\varphi), l_2) = \mathsf{run}(\mathsf{history}_M(\tau_1), l_2) * \delta(\theta_p) = l_2' * \delta(\theta_p)$. Thus, the required holds for $l_2'' = l_2' * \delta(\theta_p)$.
- Rule (14) such that $\varphi = (t, \mathsf{ret}\ m(\theta_q))$, where $\Gamma(m) = (\_, q)$ and $\theta_q \in q_t$. Then $l_1'' = l_1' \setminus \delta(\theta_q)$ and $\theta' = \theta * \theta_q$. Since $\mathsf{history}_M(\tau)$ is well-balanced from l_2, so is $\mathsf{history}_M(\tau_1\varphi)$. We thus have $\mathsf{run}(\mathsf{history}_M(\tau_1\varphi), l_2) = \mathsf{run}(\mathsf{history}_M(\tau_1), l_2) \setminus \delta(\theta_q) = l_2' \setminus \delta(\theta_q)$, which is defined. Besides, $l_2' \setminus \delta(\theta_q) \preceq l_1' \setminus \delta(\theta_q)$, so the required holds for $l_2'' = l_2' \setminus \delta(\theta_q)$.
- Rule (15) such that $\varphi = (t, \mathsf{call}\ m(\theta_p))$, where $(\Gamma' - \Gamma)(m) = (p, \_)$ and $\theta_p \in p_t$. Then $l_1'' = l_1'$, $\theta' = \theta * \theta_p$ and $\{\theta\} * l_1' * \{\theta_p\} \neq \emptyset$. Since $l_2' \preceq l_1'$, $\{\theta\} * l_2' * \{\theta_p\} \neq \emptyset$. Besides, $\mathsf{run}(\mathsf{history}_M(\tau_1\varphi), l_2) = \mathsf{run}(\mathsf{history}_M(\tau_1), l_2) = l_2'$. Then, the required holds for $l_2'' = l_2'$.
- Rule (18) such that $\varphi = (t, \mathsf{call}\ m(\theta_p))$, where $(\Gamma' \cap \Gamma)(m) = (p, \_)$ and $\theta_p \in p_t$. Then $l_1'' = l_1' * \delta(\theta_p)$, $\theta' = \theta$ and $\{\theta\} * l_1' * \{\theta_p\} \neq \emptyset$. Since $l_2' \preceq l_1'$, $\{\theta\} * l_2' * \{\theta_p\} \neq \emptyset$ and $l_2' * \delta(\theta_p) \preceq l_1''$. Besides, $\mathsf{run}(\mathsf{history}_M(\tau_1\varphi), l_2) = \mathsf{run}(\mathsf{history}_M(\tau_1), l_2) * \delta(\theta_p) = l_2' * \delta(\theta_p)$. Then, the required holds for $l_2'' = l_2' * \delta(\theta_p)$.
- Rule (19) such that $\varphi = (t, \mathsf{ret}\ m(\theta_q))$, where $(\Gamma' \cap \Gamma)(m) = (\_, q)$ and $\theta_q \in q_t$. Then $l_1'' = l_1' \setminus \delta(\theta_q)$ and $\theta' = \theta$. Since $l_2' \preceq l_1'$, $l_2' \setminus \delta(\theta_q) \preceq l_1''$. Besides, $\mathsf{run}(\mathsf{history}_M(\tau_1\varphi), l_2) = \mathsf{run}(\mathsf{history}_M(\tau_1), l_2) \setminus \delta(\theta_q) = l_2' \setminus \delta(\theta_q)$, which is defined. Thus, the required holds for $l_2'' = l_2' \setminus \delta(\theta_q)$.
- It is easy to check that in all other cases we have $l_1' = l_1''$, $\mathsf{run}(\mathsf{history}_M(\tau\varphi), l_2) = \mathsf{run}(\mathsf{history}_M(\tau), l_2) = l_2'$ and the required holds for $l_2'' = l_2'$.

□

B.5 Proof of Lemma 14

Assume $(\theta_0, \delta(\theta_L) * l_0, \eta) \in [\![\mathcal{P}]\!]$ and $(\theta_L, l_0, \xi) \in [\![\mathcal{L}]\!]$ such that $\mathsf{history}_{\mathsf{dom}(\Gamma_1)}(\eta) = \mathsf{history}_{\mathsf{dom}(\Gamma_1)}(\xi)$. Then there exist an η-labelled derivation using $\longrightarrow_{\mathcal{P}}$ and a $x i$-labelled derivation using $\longrightarrow_{\mathcal{L}}$, starting from initial configurations $\varsigma_0^1 \in \mathsf{Config}$ and $\varsigma_0^2 \in \mathsf{Config}$, 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 $(\theta_0 * \theta_L, l_0, \tau) \in [\![\mathcal{P}(\mathcal{L})]\!]$ together with its derivation using $\longrightarrow_{\mathcal{P}}$. We first build a series of finite 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 $\varsigma_0^1 * \varsigma_0^2$ 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 finite prefixes η_1 and ξ_1 of η and ξ such that

$$\begin{aligned}&\mathsf{history}_{\mathsf{dom}(\Gamma_1)}(\eta_1) = \mathsf{history}_{\mathsf{dom}(\Gamma_1)}(\xi_1)\\ \wedge\ &\mathsf{client}_{\mathsf{dom}(\Gamma_1)}(\tau_i) = \mathsf{ierase}_{(\mathsf{dom}(\Gamma_1)-\mathsf{dom}(\Gamma_0))\cap\mathsf{dom}(\Gamma_2)}(\eta_1)\\ \wedge\ &\mathsf{lib}_{\mathsf{dom}(\Gamma_1)}(\tau_i) = \mathsf{erase}_{\mathsf{dom}(\Gamma_1)-\mathsf{dom}(\Gamma_0)}(\xi_1)\end{aligned}$$

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

$$\varsigma_0^1 * \varsigma_0^2 \xrightarrow{\tau_i}{}^*_{\mathcal{P}} \varsigma_i^1 * \varsigma_i^2 \wedge \varsigma_0^1 \xrightarrow{\eta_1}{}^*_{\mathcal{P}} \varsigma_i^1 \wedge \varsigma_0^2 \xrightarrow{\xi_1}{}^*_{\mathcal{L}} \varsigma_i^2.$$

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 that $\eta = \eta_1$ 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:

$$\varsigma_i^1 \xrightarrow{\varphi_1}_{\mathcal{P}} \varsigma^1 \quad \wedge \quad \varsigma_i^2 \xrightarrow{\varphi_2}_{\mathcal{L}} \varsigma^2,$$

where $\varsigma^1, \varsigma^2 \in \mathsf{Config} - \{\top\}$.

Let $\varphi_1 = (t_1, \_)$, $\varphi_2 = (t_2, \_)$ and

$$\sigma_i^1 = \varsigma_i^1|_t \ \wedge\ \sigma^1 = \varsigma^1|_t \ \wedge\ \sigma_i^2 = \varsigma_i^2|_t \ \wedge\ \sigma^2 = \varsigma^2|_t.$$

Then $\sigma_i^1 * \sigma_i^2$ is defined and

$$\sigma_i^1 \xrightarrow{\varphi}_{t, \mathcal{P}(\mathcal{L})} \sigma^1 \ \wedge\ \sigma_i^2 \xrightarrow{\varphi}_{t, \mathcal{P}(\mathcal{L})} \sigma^2.$$

We now make a case-split on types of actions φ_1 and φ_2 and which of rules (8)–(19) are used to derive them.

- $\varphi_1, \varphi_2 \in \mathsf{ECallAct}_{\mathsf{dom}(\Gamma_1)}$ derived using rules (12) and (15), respectively and such that φ_2 is an outermost call to methods from $\mathsf{dom}(\Gamma_1)$ in ξ. Then φ_1 is an outermost call to methods from $\mathsf{dom}(\Gamma_1)$ in η. In this case $\varphi_1 = \varphi_2$, because $\mathsf{history}_{\mathsf{dom}(\Gamma_1)}(\eta_1)$ and $\mathsf{history}_{\mathsf{dom}(\Gamma_1)}(\xi_1)$ are the same by our assumption, so that their same-length prefixes $\mathsf{history}_{\mathsf{dom}(\Gamma_1)}(\tau_i)\varphi_1$ and $\mathsf{history}_{\mathsf{dom}(\Gamma_1)}(\tau_i)\varphi_2$ should be identical. Assume $\varphi_1 = (t, \mathsf{call}\ m(\theta_p))$ for $\Gamma(m) = (p, \_)$ and $\theta_p \in p_t$. Then, there exist $v, v' \in \mathsf{Node}$, $\alpha \in \mathsf{Pos}$, $l \in \mathsf{Foot}$,

$\theta_1, \theta_2 \in \mathsf{State}$, such that (v, m, v') is in the control-flow relation of $\mathcal{P}$, $\theta_1 = \theta_1' * \theta_p$ and

$$\begin{aligned} & \sigma_i^1 = (\alpha v, \theta_1, \delta(\theta_2) * l) \\ \wedge\ & \sigma^1 = (\alpha v' v_m, \theta_1', \delta(\theta_2) * l * \delta(\theta_p)) \\ \wedge\ & \sigma_i^2 = (v^t_{\mathsf{mgc}}, \theta_2, l) \\ \wedge\ & \sigma^2 = (v^t_{\mathsf{mgc}}\mathsf{start}_m, \theta_2 * \theta_p, l) \\ \wedge\ & \sigma_i^1 * \sigma_i^2 = (\alpha v, \theta_1 * \theta_2, l) \quad \wedge \quad \theta_1 = \theta_1' * \theta_p. \end{aligned}$$

In this case

$$\sigma^1 * \sigma^2 = (\alpha v' \mathsf{start}_m, \theta_1 * \theta_2, l),$$

is defined and

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

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

$$\varsigma_i^1 * \varsigma_i^2 \xrightarrow{(t, \mathsf{call}\ m)}_{\mathcal{P}(\mathcal{L})} \varsigma^1 * \varsigma^2.$$

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

$$\varsigma_0^1 * \varsigma_0^2 \xrightarrow{\tau_i}{}^*_{\mathcal{P}(\mathcal{L})} \varsigma_i^1 * \varsigma_i^2 \xrightarrow{(t, \mathsf{call}\ m)}_{\mathcal{P}(\mathcal{L})} \varsigma^1 * \varsigma^2.$$

- $\varphi_1, \varphi_2 \in \mathsf{ECallAct}_{\mathsf{dom}(\Gamma_1)}$ such that φ_2 is an outermost call to methods from $\mathsf{dom}(\Gamma_1)$ in ξ and the actions are derived using rules (18) and (18), (12) and (18), or (18) and (15), respectively. Then φ_1 is an outermost call to methods from $\mathsf{dom}(\Gamma_1)$ in η. These cases are handled analogously to the previous one.
- $\varphi_1, \varphi_2 \in \mathsf{ERetAct}_{\mathsf{dom}(\Gamma_1)}$ such that φ_2 is an outermost return from methods in $\mathsf{dom}(\Gamma_1)$ in ξ and the actions are derived using rules (14) and (16), (19) and (19), (14) and (19), or (19) and (16), respectively. Then φ_1 is an outermost return from methods in $\mathsf{dom}(\Gamma_1)$ in η. This case is handled analogously to the previous one.
- $\varphi_1 \in \mathsf{ERetAct}_{\mathsf{dom}(\Gamma_1)}$ and $\varphi_2 \in \mathsf{PAct}$, such that φ_2 is derived using (8) and the actions are performed by the same thread. Then φ_1 is an outermost return from methods in $\mathsf{dom}(\Gamma_1)$ in η. In this case for some $\theta_1, \theta_2, \theta_2' \in \mathsf{State}$, $v, v' \in \mathsf{Node}$, $\alpha, \beta \in \mathsf{Pos}$ and $l \in \mathsf{Foot}$, we have that (v, c, v') is in the control-flow relation of $\mathcal{P}(\mathcal{L})$ and

$$\begin{aligned} & \sigma_i^1 = (\alpha v_m, \theta_1, \delta(\theta_2) * l) \ \wedge \\ & \sigma_i^2 = (v^t_{\mathsf{mgc}}\beta v, \theta_2, l) \ \wedge \\ & \sigma^2 = (v^t_{\mathsf{mgc}}\beta v', \theta_2', l) \ \wedge \ \theta_2' \in f_c(\theta_2) \ \wedge \\ & \sigma_i^1 * \sigma_i^2 = (\alpha\beta v, \theta_1 * \theta_2, l). \end{aligned}$$

Since $\theta_1 * \theta_2$ is defined, by (3), $\theta_2' \in f_c(\theta_2)$ implies that $\theta_1 * \theta_2'$ is defined as well. Then

$$\sigma_i^1 * \sigma^2 = (\alpha\beta v', \theta_1 * \theta_2', l).$$

is defined. From (2) we have

$$\theta_1 * \theta_2' \in \{\theta_1\} * f_c(\theta_2) = f_c(\theta_1 * \theta_2).$$

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

$$\varsigma_0^1 * \varsigma_0^2 \xrightarrow{\tau_i}{}^*_{\mathcal{P}(\mathcal{L})} \varsigma_i^1 * \varsigma_i^2 \xrightarrow{\varphi_2}_{\mathcal{P}(\mathcal{L})} \varsigma_i^1 * \varsigma^2.$$

- $\varphi_1 \in \mathsf{ERetAct}_{\mathsf{dom}(\Gamma_1)}$ and $\varphi_2 \in \mathsf{CallRetAct} \cup \mathsf{ECallRetAct}_{\mathsf{dom}(\Gamma_0)}$ such that the actions are executed by the same thread and φ_2 is not an outermost call to or return from a method in $\mathsf{dom}(\Gamma_1)$ in ξ. Then φ_1 is an outermost return from methods in $\mathsf{dom}(\Gamma_1)$ in η. This case is handled similarly to the previous one.
- $\varphi_1 \in \mathsf{ECallAct}_{\mathsf{dom}(\Gamma_2 - \Gamma_1)}$ and $\varphi_2 \in \mathsf{ECallAct}_{\mathsf{dom}(\Gamma_1)}$, such that the actions are performed by the same thread and φ_1 is derived using rule (12). Then φ_2 is an outermost call to methods from $\mathsf{dom}(\Gamma_1)$ in η. Let $\varphi_1 = (t, \mathsf{call}\ m(\theta_p))$, $(\Gamma_2 - \Gamma_1)(m) = (p, \_)$, $\theta_p \in p_t$. In this case for some $\theta_1, \theta_2 \in \mathsf{State}$, $\theta_p \in p$, $v, v' \in \mathsf{Node}$, $\alpha \in \mathsf{Pos}$ and $l \in \mathsf{Foot}$, we have

$$\begin{aligned} & \sigma_i^1 = (w^t_{\mathsf{mgc}}, \theta_1, \delta(\theta_2) * l) \ \wedge \\ & \sigma_i^2 = (v^t_{\mathsf{mgc}}, \theta_2, l) \ \wedge \\ & \sigma^1 = (w^t_{\mathsf{mgc}}\mathsf{start}_m, \theta_1 * \theta_p, l) \ \wedge \\ & \sigma_i^1 * \sigma_i^2 = (w^t_{\mathsf{mgc}}, \theta_1 * \theta_2, l), \end{aligned}$$

where $\{\theta_1\} * \delta(\theta_2) * l * \{\theta_p\} \neq \emptyset$. Hence,

$$\sigma^1 * \sigma_i^2 = (w^t_{\mathsf{mgc}}\mathsf{start}_m, \theta_1 * \theta_2 * \theta_p, l).$$

is defined (note that this relies crucially on the requirement that the state θ_p allocated in rule (12) be compatible with the footprint of the imported libraries). Hence, $\sigma_i^1 * \sigma_i^2 \xrightarrow{\varphi_1}_{t, \mathcal{P}(\mathcal{L})} \sigma^1 * \sigma_i^2$. Then $\varsigma^1 * \varsigma_i^2$ is defined and $\varsigma_i^1 * \varsigma_i^2 \xrightarrow{\varphi_1}_{\mathcal{P}(\mathcal{L})} \varsigma^1 * \varsigma_i^2$. The desired τ_{i+1} is $\tau_i\varphi_1$, and its derivation is:

$$\varsigma_0^1 * \varsigma_0^2 \xrightarrow{\tau_i}{}^*_{\mathcal{P}(\mathcal{L})} \varsigma_i^1 * \varsigma_i^2 \xrightarrow{\varphi_1}_{\mathcal{P}(\mathcal{L})} \varsigma^1 * \varsigma_i^2.$$

- $\varphi_1 \in \mathsf{PAct} \cup \mathsf{CallRetAct}$ and $\varphi_2 \in \mathsf{ECallAct}_{\mathsf{dom}(\Gamma_1)}$, such that the actions are performed by the same thread. Then φ_2 is an outermost call to methods from $\mathsf{dom}(\Gamma_1)$ in η. This case is handled similarly to the previous one.
- $\varphi_1 \in \mathsf{PAct} \cup \mathsf{CallRetAct} \cup \mathsf{ECallRetAct}_{\mathsf{dom}(\Gamma_2 - \Gamma_1)}$ and $\varphi_2 \in \mathsf{PAct} \cup \mathsf{CallRetAct} \cup \mathsf{ECallRetAct}_{\mathsf{dom}(\Gamma_0)}$, such that the actions are executed by different threads and φ_2 is not an outermost call to or return from a method in $\mathsf{dom}(\Gamma_1)$ in ξ. As before, we can show that

$$\varsigma_i^1 * \varsigma_i^2 \xrightarrow{\varphi_1}_{\mathcal{P}(\mathcal{L})} \varsigma^1 * \varsigma_i^2 \quad \wedge \quad \varsigma^1 * \varsigma_i^2 \xrightarrow{\varphi_2}_{\mathcal{P}(\mathcal{L})} \varsigma^1 * \varsigma^2.$$

This gives the extension of the given computation for τ_i by two further steps:

$$\varsigma_0^1 * \varsigma_0^2 \xrightarrow{\tau_i\varphi_1\varphi_2}{}^*_{\mathcal{P}(\mathcal{L})} \varsigma^1 * \varsigma^2.$$

It follows that the trace $\tau_i\varphi_1\varphi_2$ is the desired τ_{i+1}.

It is easy to check that all the other cases are impossible.

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

$$\mathsf{client}_{\mathsf{dom}(\Gamma_1)}(\tau) = \mathsf{ierase}_{(\mathsf{dom}(\Gamma_1) - \mathsf{dom}(\Gamma_0)) \cap \mathsf{dom}(\Gamma_2)}(\eta)$$

and

$$\mathsf{erase}_{\mathsf{dom}(\Gamma_1) - \mathsf{dom}(\Gamma_0)}(\mathsf{lib}_{\mathsf{dom}(\Gamma_1)}(\tau)) = \mathsf{erase}_{\mathsf{dom}(\Gamma_1) - \mathsf{dom}(\Gamma_0)}(\xi).$$

□

B.6 Proof of Theorem 3

Consider θ_0, θ_L and l_0 such that $(\theta_L, l_0) \in \mathcal{I}_L$ $(\theta_0, \delta(\theta_L) * l_0) \in \mathcal{I}_P$. By contradiction, assume there is an unsafe execution of $\mathcal{P}(\mathcal{L})$: for some trace τ, $\varphi \in \mathsf{Act}$, $\mathsf{pc} \in \mathsf{ThreadID} \rightharpoonup \mathsf{Pos}$, $\theta \in \mathsf{State}$, we have

$$(\mathsf{pc}_0, \theta_0 * \theta_L, l_0, \mathsf{ThreadID}) \xrightarrow{\tau}{}^*_{\mathcal{P}(\mathcal{L})} (\mathsf{pc}, \theta, l, \kappa) \xrightarrow{\varphi}_{\mathcal{P}(\mathcal{L})} \top.$$

Let $\varphi = (t, \_)$. The prefix τ of the faulting trace is safe. From the proof of Lemma 12, for some η and ξ such that $(\theta_0, \delta(\theta_L) * l_0, \eta) \in [\![\mathcal{P}]\!]$ and $(\theta_L, l_0, \xi) \in [\![\mathcal{L}]\!]$, and $\theta_1, \theta_2 \in \mathsf{State}$ such that $\theta = \theta_1 * \theta_2$, we have

$$\begin{aligned} & (\mathsf{pc}_0, \theta_0, \delta(\theta_L), \mathsf{ThreadID}) \xrightarrow{\eta}{}^*_{\mathcal{P}} \\ & \qquad (\mathsf{client}_{\mathsf{dom}(\Gamma_1)}(\mathsf{pc}), \theta_1, \delta(\theta_2) * l, \mathsf{client}_{\mathsf{dom}(\Gamma_1), \mathsf{pc}(t), t}(\kappa)) \end{aligned}$$

and

$$\begin{aligned} & (\mathsf{pc}_0, \theta_L, \delta(l_0), \mathsf{ThreadID}) \xrightarrow{\xi}{}^*_{\mathcal{L}} \\ & \qquad (\mathsf{lib}_{\mathsf{dom}(\Gamma_1)}(\mathsf{pc}), \theta_2, l, \mathsf{lib}_{\mathsf{dom}(\Gamma_1), \mathsf{pc}(t), t}(\kappa)). \end{aligned}$$

We now show that either $\mathcal{P}$ or $\mathcal{L}$ is unsafe by considering every rule of the operational semantics that may produce the faulting transition φ.

- Rule (9) such that $\varphi = (t, c) \in \mathsf{PAct}$ and $f_c^t(\theta) = \top$. By (2) this implies $f_c^t(\theta_1) = f_c^t(\theta_2) = \top$. Besides, for some $v, v' \in \mathsf{Node}$, (v, c, v') is in the control-flow relation of $\mathcal{P}(\mathcal{L})$ and $\mathsf{pc}(t) = v$ ends with v. If v belongs to the code of $\mathcal{P}$, then

$$(\mathsf{client}_{\mathsf{dom}(\Gamma_1)}(\mathsf{pc}), \theta_1, \delta(\theta_2) * l, \mathsf{client}_{\mathsf{dom}(\Gamma_1), \mathsf{pc}(t), t}(\kappa)) \xrightarrow{\varphi} \top;$$

if it belongs to $\mathcal{L}$, then

$$(\mathsf{lib}_{\mathsf{dom}(\Gamma_1)}(\mathsf{pc}), \theta_2, l, \mathsf{lib}_{\mathsf{dom}(\Gamma_1),\mathsf{pc}(t),t}(\kappa)) \xrightarrow{\varphi} \top.$$

In both cases either $\mathcal{P}$ or $\mathcal{L}$ is unsafe.

- Rule (13) such that $\varphi = (t, \mathsf{call}\ m(\epsilon))$. Let $\Gamma_0(m) = (p, \_)$. Then for some $v, v' \in \mathsf{Node}$, (v, m, v') is in the control-flow relation of $\mathcal{P}(\mathcal{L})$, $\mathsf{pc}(t)$ ends with v, and $\theta \notin \mathsf{State} * p_t$. The latter implies $\theta_1, \theta_2 \notin \mathsf{State} * p_t$. Like in the previous case, this means that either $\mathcal{P}$ or $\mathcal{L}$ is unsafe.
- Rule (17) such that $\varphi = (t, \mathsf{ret}\ m(\epsilon))$ and $m \in \mathsf{dom}(\Gamma_2) - \mathsf{dom}(\Gamma_1)$. Let $(\Gamma_2 \cap \Gamma_1 - \Gamma_0)(m) = (\_, q)$, then $\mathsf{pc}(t) = v^t_{\mathsf{mgc}}\mathsf{end}_m$ and $\theta \notin \mathsf{State} * q_t$. The latter implies $\theta_1 \notin \mathsf{State} * q_t$. But then $\mathcal{P}$ is unsafe.
- Rule (17) such that $\varphi = (t, \mathsf{ret}\ m(\epsilon))$ and $m \in (\mathsf{dom}(\Gamma_2) \cap \mathsf{dom}(\Gamma_1)) - \mathsf{dom}(\Gamma_0)$. Let $(\Gamma_2 \cap \Gamma_1 - \Gamma_0)(m) = (\_, q)$, then $\mathsf{pc}(t) = v^t_{\mathsf{mgc}}\mathsf{end}_m$ and $\theta \notin \mathsf{State} * q_t$. The latter implies $\theta_2 \notin \mathsf{State} * q_t$. But then $\mathcal{L}$ is unsafe.

□

C. Logic for safety

In this section we review an existing program logic that can be used to reason about open programs of Section 3. In Appendix D we extend the logic presented here for establishing the notion of linearizability proposed in Section 4.

Proving the safety of open programs is convenient in separation logics [29], 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 [32], which combines rely-guarantee (aka assume-guarantee) reasoning [19, 27] with separation logic [29].

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

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

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

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

$$\begin{array}{lcl} \theta, \mathbf{i} \models X & \iff & \theta = \mathbf{i}(X) \\ \theta, \mathbf{i} \models \exists X.\, p & \iff & \exists \theta' \in \mathsf{State}.\, (\theta, \mathbf{i}[X : \theta'] \models p) \\ \theta, \mathbf{i} \models \exists x.\, p & \iff & \exists v \in \mathbb{Z}.\, (\theta, \mathbf{i}[x : v] \models p) \\ \theta, \mathbf{i} \models \mathsf{emp} & \iff & \theta = \epsilon \\ \theta, \mathbf{i} \models p * q & \iff & \exists \theta_0, \theta_1.\, (\theta_0 * \theta_1)\!\downarrow \wedge\, \theta_0 * \theta_1 = \theta \wedge \\ & & \theta_0, \mathbf{i} \models p \wedge \theta_1, \mathbf{i} \models q \\ \theta, \mathbf{i} \models p \mathrel{-\!\!*} q & \iff & \forall \theta'.\, ((\theta * \theta')\!\downarrow \wedge\, (\theta', \mathbf{i} \models p)) \Rightarrow \\ & & (\theta * \theta', \mathbf{i} \models q) \end{array}$$

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 extensively in our extension of the logic for reasoning about linearizability (Appendix D). 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 $[\![p]\!]_t$.

The above assertion language can be extended as needed when we consider particular instatiations of State. 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, $[\![p]\!] = \{(\theta, \mathbf{i}) \mid (\theta, \mathbf{i} \models p)\}$.

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 $\mathsf{State} \times \mathsf{State} \times \mathsf{LInt}$. 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:

$$P, Q ::= p \mid \boxed{p} \mid P * Q \mid P \wedge Q \mid P \vee Q$$

with the following semantics:

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

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 $\mathsf{State} \times \mathsf{State}$. Such 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 a binary 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 with `atomic { C }` a block of code C considered as one atomic command. While proving a program, we assume that atomic method declarations in it are replaced with ordinary ones, but with method implementations wrapped into `atomic` blocks.

The judgements of the logic have the form

$$\Gamma \mid \Delta \mid R, G \vdash^w_t \{P\}\, C\, \{Q\}.$$

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 3, where pre- and postconditions are syntactic assertions denoting parameterised predicates. We require that the assertions p be insensitive to interpretations of logical variables in the following sense: $\forall (\theta, \mathbf{i}) \in p.\, \forall \mathbf{i}'.\, (\theta, \mathbf{i}') \in p$. We also require that the assertions p 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 3). The context Δ

is a finite map from methods to rely-guarantee specifications of the form $R', G' \rhd \{P'\}\, m\, \{Q'\}$. The context Δ provides specifications of methods that are implemented in the program considered. Note that a specification of a method in Δ includes not only pre- and postconditions, but also rely and guarantee conditions the method admits. The domains of Γ and Δ are required to be disjoint.

Informally, our judgement $\Gamma \mid \Delta \mid R, G \vdash_t^w \{P\}\, C\, \{Q\}$ 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 and Δ. 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

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

for an open program $\mathcal{C}$ with a ground client, which may call methods undefined but specified in Γ or Δ.

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. Atomic commands of the latter kind are annotated with actions $p \rightsquigarrow q$, as in $\texttt{atomic}_{p \rightsquigarrow q}\ \{\ C\ \}$, which determine how the command treats the shared state (see below). These annotations are a part of proofs in our logic.

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, we require that assertions P, Q in a judgement $\Gamma \mid \Delta \mid R, G \vdash_t^w \{P\}\, C\, \{Q\}$ 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

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

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 $\mathsf{State} \times \mathsf{LInt}$:

$$f_c^t(\theta, \mathbf{i}) = f_c^t(\theta) \times \{\mathbf{i}\}, \tag{30}$$

if $f_c^t(\theta) \neq \top$; $f_c^t(\theta, \mathbf{i}) = \top$, otherwise. When particular State 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.

The ATOMICOUT rule handles commands accesssing the shared state, according to the corresponding annotation $p_0 \rightsquigarrow q_0$: 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 and the environment did not interfere. To model the latter the rely is replaced by $\emptyset$. 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. The ATOMICIN rule just ignores nested atomic blocks.

The consequence rule (CONSEQ) allows strenthening 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 the current rely R in case C contains commands changing the shared state.

The CALL1 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 idendifier

Figure 5. Proof rules of RGSep

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

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

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

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

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

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

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

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

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

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

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

$$\frac{}{\Gamma, \{p\}\, m\, \{q\} \mid \Delta \mid R, G \vdash_t^w \{P * p_t\}\, m\, \{P * q_t\}}\ \textsc{Call1}$$

$$\frac{}{\Gamma \mid \Delta, (R, G \rhd \{P\}\, m\, \{Q\}) \mid R, G \vdash_t^w \{P\}\, m\, \{Q\}}\ \textsc{Call2}$$

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

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

$$\frac{\begin{array}{l}\Gamma \mid \Delta, \Delta' \vdash \{P\}\, \mathcal{C}\, \{Q\} \\ \forall (R, G \rhd \{P_m\}\, m\, \{Q_m\}) \in \Delta'.\ \forall t \in \mathsf{ThreadID}. \\ \quad m \in M \text{ and } (\Gamma \mid \Delta, \Delta' \mid R, G \vdash_t^w \{P_m\}\, C_m\, \{Q_m\})\end{array}}{\Gamma \mid \Delta \vdash \{P\}\, \mathsf{let}\, \{m = C_m \mid m \in M\}\, \mathsf{in}\, \mathcal{C}\, \{Q\}}\ \textsc{Library}$$

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 CALL2 axiom allows using a method specification from Δ with the rely and the guarantee matching the current ones.

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. 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. Finally, the LIBRARY rule adapts the standard procedure declaration rule to our setting and is used to reason about library declarations.

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

Theorem 25 (Soundness—client). *Consider a program* $\Gamma \vdash \mathcal{C} : \emptyset$ *with an initial condition* $\mathcal{I}$*. Assume an assertion* P *is such that*

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

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

Theorem 26 (Soundness—library). *Consider a program* $\Gamma \vdash \mathcal{L} : \Gamma'$ *with an initial condition* $\mathcal{I}$*. Assume a guarantee* G*, an assertion* inv *and a context* Δ *such that*

- $\forall (\theta, l) \in \mathcal{I}.\, \exists (\theta, \mathbf{i}) \in [\![\mathsf{inv}]\!]$*;*
- $\boxed{\mathsf{inv}}$ *is stable under* G*;*
- *for all* $(R', G' \rhd \{P_m\}\, m\, \{Q_m\}) \in \Delta$ *and* $t \in \mathsf{ThreadID}$

$$\Gamma \mid \Delta \mid R', G' \vdash_t^w \{P_m\}\, C_m\, \{Q_m\};$$

- *for all* $\{p^m\}\, m\, \{q^m\} \in \Gamma' - \Gamma$ *and* $t \in \mathsf{ThreadID}$

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

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

The proofs are identical to the soundness proof of RGSep [32].

D. Logic for linearizability with ownership transfer

We now extend the logic presented in Appendix C to reasoning about our notion of linearizability (Section 4). The logic we present here generalises the method of proving linearizability using linearization points [1, 16, 32] to the setting with ownership transfer.

Consider the following program $\Gamma \vdash \mathcal{L} : \Gamma'$ with an initial condition $\mathcal{I}$:

$$\begin{array}{rcl} \Gamma' & = & \lambda m \in M.\, \{p^m\}\, m\, \{q^m\}, \\ \mathcal{L} & = & \mathsf{let}\ \{m = C_m \mid m \in M_1\}\ \mathsf{in} \\ & & \mathsf{let}\ \{m = C_m \mid m \in M_2\}\ \mathsf{in} \\ & & \ldots \\ & & \mathsf{let}\ \{m = C_m \mid m \in M_k\}\ \mathsf{in} \\ & & [-] \end{array}$$

for some sets M, M_i of methods such that $M \subseteq \bigcup_{i=1..k} M_i$. 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 $(\mathcal{L}, \mathcal{I}) \sqsubseteq (\mathcal{L}', \mathcal{I}')$, where

$$\begin{array}{rcl} \mathcal{L}' & = & \mathsf{let}\ \{\mathsf{atomic}\ m = (\mathsf{skip}^*; C_m^a; \mathsf{skip}^*) \mid m \in M_1\}\ \mathsf{in} \\ & & \mathsf{let}\ \{\mathsf{atomic}\ m = (\mathsf{skip}^*; C_m^a; \mathsf{skip}^*) \mid m \in M_2\}\ \mathsf{in} \\ & & \ldots \\ & & \mathsf{let}\ \{\mathsf{atomic}\ m = (\mathsf{skip}^*; C_m^a; \mathsf{skip}^*) \mid m \in M_k\}\ \mathsf{in} \\ & & [-]. \end{array}$$

for some commands C_m^a, $m \in \bigcup_{i=1..k} M_i$.

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 [14].

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.

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

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

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

$$p_r, q_r \ ::= \ \ldots \mid p \mid \lfloor p \rfloor$$

of relational assertions, where $\ldots$ duplicates the grammar of the assertions $p, q, \ldots$ denoting subsets of $\mathsf{State} \times \mathsf{LInt}$, but now with p, q replaced by p_r, q_r. The assertion $\lfloor p \rfloor$ denotes a state of the abstract implementation satisfying p. Note that the resulting grammar for p_r disallows nested $\lfloor \cdot \rfloor$ operators.

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

$$\begin{array}{rcl} \theta_c, \theta_a, \mathbf{i} \models p & \iff & (\theta_c, \mathbf{i} \models p) \wedge \theta_a = \epsilon \\ \theta_c, \theta_a, \mathbf{i} \models \lfloor p \rfloor & \iff & (\theta_a, \mathbf{i} \models p) \wedge \theta_c = \epsilon \end{array}$$

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

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

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 is given in Figure 6. The semantic 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 $\mathsf{State}^2 \times \mathsf{State}^2$.

The judgements of the new proof system have the form

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

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 transformers f_c^t defining the semantics of primitive commands c to $\mathsf{State} \times \mathsf{State}$ in the following two ways: for all $\theta, \theta' \in \mathsf{State}$

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

if f_c^t on the corresponding single state is distinct from $\top$; 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 $\mathsf{State} \times \mathsf{State} \times \mathsf{LInt}$ similarly to (30). We now replace the PRIM axiom with two corresponding variants:

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

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 $\vdash_t^j$. We only need to change the notion of stability: P_r is stable under R_r when

$$\forall((\theta_c, \theta_a), (\theta_1^c, \theta_1^a), \theta, \nu, \mathbf{i}) \in [\![P_r]\!].\, \forall \theta_2^c, \theta_2^a \in \mathsf{State}.$$
$$((\theta_1^c, \theta_1^a), (\theta_2^c, \theta_2^a)) \in [\![R_r]\!] \Rightarrow ((\theta_c, \theta_a), (\theta_2^c, \theta_2^a), \theta, \nu, \mathbf{i}) \in [\![P_r]\!].$$

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

The proof method for linearizability builds on the technique for proving the safety of library implementations described in Theorem 26. To prove $(\mathcal{L}, \mathcal{I}) \sqsubseteq (\mathcal{L}', \mathcal{I}')$, we require a guarantee G_r, an assertion inv_r, and a context Δ such that

- $\forall (\theta, l) \in \mathcal{I}, (\theta', l') \in \mathcal{I}'.\, \exists \mathbf{i}.\, (\theta, \theta', \mathbf{i}) \in [\![\mathsf{inv}_r]\!]$;
- $\boxed{\mathsf{inv}_r}$ is stable under G_r;
- for all $(R_r', G_r' \rhd \{P_r\}\, m\, \{Q_r\}) \in \Delta$ and $t \in \mathsf{ThreadID}$

$$\Gamma \mid \Delta \mid R_r', G_r' \vdash_t \{P_r\}\, C_m\, \{Q_r\}.$$

For an implementation C_m of a method m in the domain of $\Gamma' - \Gamma$, let $\tilde{C}_m$ be C_m with some of commands C inside atomic blocks replaced by $C; (\!|C_m^a|\!)$. Here we mark with $(\!|\cdot|\!)$ the code of the abstract implementation, which is treated specially by the proof system. The placement of $(\!|C_m^a|\!)$ thus fixes linearization points inside the code of C_m. For every thread t and $\{p^m\}\ m\ \{q^m\} \in \Gamma' - \Gamma$, we require a derivation of the following judgement:

$$\begin{array}{l}\Gamma \mid \Delta \mid G_r, G_r \vdash_t^{\mathsf{conc}} \\ \quad \{\exists X.\, \mathsf{Pre}(X) \wedge p_t^m(X) \wedge X * \boxed{\mathsf{inv}}\} \\ \quad \tilde{C}_m \\ \quad \{\exists Y.\, \mathsf{Post}(Y) \wedge q_t^m(Y) \wedge Y * \boxed{\mathsf{inv}}\}\end{array} \tag{31}$$

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

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

The rationale behind (31) and LINPOINT is as follows. As noted above, in the method of linearization points 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 (31), $\mathsf{Pre}(X)$ 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 (31) 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 exactly 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 27 (Soundness). *Under the above conditions, $\mathcal{L}$ and $\mathcal{L}'$ are safe for $\mathcal{I}$ and $\mathcal{I}'$, respectively, and $(\mathcal{L}, \mathcal{I}) \sqsubseteq (\mathcal{L}', \mathcal{I}')$.*

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.

E. Proof of Michael and Scott's queue

Non-blocking queue with a memory allocator. In Figure 7 we present the concurrent queue implementation due to Michael and Scott [21] we refered to in Section 6. For readability, in examples we use the full C language instead of the minimalistic one for which we formalise our results. A client using the implementation can call several `enqueue`, `dequeue` or `isEmpty` operations concurrently. The queue is non-blocking, i.e., implemented with compare-and-swap operations (CAS) instead of locks. CAS takes three arguments: a memory address `addr`, an expected value `v1`, and a new value `v2`. It atomically reads the memory address and updates it with the new value when the address contains the expected value; otherwise, it does nothing. In C syntax this might be written as follows:

```
atomic {
  if (*addr==v1) { *addr=v2; return 1; }
  else { return 0; }
}
```

In most architectures an efficient CAS (or an equivalent operation) is provided natively by the processor.

Like most concurrent algorithms with explicit memory management, the queue algorithm uses a custom memory allocator to allocate `Node` structures, which Michael and Scott implement using a concurrent non-blocking stack due to Treiber [30] (Figure 8). We first explain this algorithm, as the simpler one of the two.

Memory allocator. The allocator stores the free list of memory blocks of size `sizeof(Node)` as a linked list, pointed to by the variable `Top`. The pointer to the next element of the list is stored at the beginning of each block. We assume `sizeof(Node)` $\geq$ `sizeof(Block*)` + `sizeof(unsigned)`, so that the pointer can be stored without overwriting the last counter field of the structure. As we noted in Section 6, the queue algorithm relies on this field not being changed by the memory allocator even after a node is deallocated. For brevity we omitted the initialisation code.

The operations on the list are implemented as follows. The `free` operation *(i)* reads the current value of the top-of-the-stack pointer `Top`; *(ii)* stores the read value of `Top` at the beginning of the block

Figure 7. Michael and Scott's non-blocking queue [21]

```
struct Node { NodeRef next; int val; };
struct NodeRef { Node *ptr; unsigned count; };
NodeRef head, tail;

void init() {
  head = (Node*)alloc();
  head->next = NULL;
  tail = head;
}

int enqueue(int val) {
  Node *node;
  NodeRef next, last;
  node = alloc();
  if (node == NULL) return FAIL;
  node->val = val;
  node->next.ptr = NULL;
  while (true) {
    atomic { last = tail; }
    atomic { next = last.ptr->next; }
    if (atomic { tail != last }) continue;
    if (next.ptr == NULL) {
      if (CAS(last.ptr->next, next,
              NodeRef(node, next.count+1)))
        break;
    } else
      CAS(tail, last, NodeRef(next.ptr, last.count+1));
  }
  CAS(tail, last, NodeRef(next.ptr, last.count+1));
  return SUCCESS;
}

int dequeue() {
  NodeRef next, first, last;
  int val;
  while (true) {
    atomic { first = head; }
    atomic { last = tail; }
    atomic { next = first.ptr->next; }
    if (atomic { head != first }) continue;
    if (first.ptr == last.ptr) {
      if (next.ptr == NULL) return EMPTY;
      CAS(tail, last, NodeRef(next.ptr, last.count+1));
    } else {
      atomic { val = next->val; }
      if (CAS(head, first,
              NodeRef(next.ptr, first.count+1)))
        break;
    }
  }
  free(first.ptr);
  return val;
}

int isEmpty() {
  NodeRef next, first, last;
  while (true) {
    atomic { first = head; }
    atomic { last = tail; }
    atomic { next = head.ptr->next; }
    if (head != first) continue;
    if (first != last) return 0;
    if (next.ptr == NULL) return 1;
    CAS(tail, last, NodeRef(next.ptr, last.count+1));
  }
}
```

`block` being deallocated; and *(iii)* atomically updates the top-of-the-stack pointer with the new value `block`. If the pointer has changed between *(i)* and *(iii)*, the CAS fails and the operation is restarted. The `alloc` operation is implemented in a similar way. Note that it returns `NULL` when the allocator is out of memory.

To avoid the *ABA* problem (Section 6), the algorithm associates a counter with the `Top` variable, incremented on every modification. This ensures that, when a CAS succeeds, the `Top` variable has not changed since it was read by the thread that executed it: the counter excludes the possibility of the variable being changed temporarily

Figure 8. A concurrent memory allocator implemented using Treiber's stack [30]

```
struct Block { Block *next; };
struct BlockRef { Block *ptr; unsigned count; };
BlockRef Top;

void free(void *block) {
  BlockRef t, x;
  do {
    atomic { t = Top; }
    (Block*)block->next = t.ptr;
    x = BlockRef((Block*)block, t.count+1);
  } while (!CAS(&Top, t, x));
}

void *alloc() {
  BlockRef t, x;
  do {
    atomic { t = Top; }
    if (t.ptr == NULL)
      return NULL;
    x = BlockRef(t.ptr->next, t.count+1);
  } while (!CAS(&Top, t, x));
  return t.ptr;
}
```

and then restored to the previous value. We assume that the size of the `NodeRef` structure is of a size such that it can be read and written to atomically. Following Michael and Scott [21], we assume that the modification counter is unbounded, which is clearly idealistic. However, a version of this algorithm with a bounded counting mechanism does get used in practice, e.g., in Java memory management[2]. In this case, a bound is picked such that an overflow will (hopefully) not occur, and a bounded counter will be equivalent to an unbounded one.

Consider an execution of the `alloc` method in which it is preempted in between reading `Top` and `t.ptr->next`. Another `alloc` method invocation might run to completion, removing the memory block `t.ptr` points to and returning a pointer to it to the client of the allocator. When the first `alloc` method wakes up, it will thus read a memory cell that is being used by the client. This does not cause a problem, since the allocator only reads the cell, but not writes to it, and free cells are never returned to the operating system. However, this means that in our proof we cannot consider the state of the allocator as being completely disjoint from the state of its client.

Non-blocking queue. We now give an explanation of the Michael and Scott's queue algorithm[3]. The algorithm in Figure 7 implements the queue as a singly-linked list with `head` and `tail` pointers. The `head` pointer always points to a dummy node, which is the first node in the list; `tail` points to either the last or second to last node in the list. The implementation calls the allocator to create new nodes in the list representing the queue. The `enqueue` operation returns `FAIL` if the allocator runs out of memory, and `SUCCESS` in all other cases. Like the allocator implementation, this algorithm uses modification counters to avoid the *ABA* problem, this time for all nodes in the queue. It relies on the fact that modification counters only increse, and are not modified by the memory allocator. As before, the queue implementation might end up reading from a memory cell freed to the allocator (but not writing to it).

The `enqueue` operation takes place in two distinct steps. Normally, the `enqueue` method creates a new node by calling the memory allocator, locates the last node in the queue, and performs the following two steps:

- executes a CAS to append the new node; and
- executes a CAS to swing the queue's `tail` from the prior last

[2] D. F. Bacon. Parallel and concurrent real-time garbage collection. Slides from a talk at a Summer School on Trends in Concurrency, 2008.

[3] M. Herlihy and N. Shavit. The Art of Multiprocessor Programming, 2008.

node to the current last node.

Because these two steps are not executed atomically, every other method call must be prepared to encounter a half-finished `enqueue` call, and to finish the job ("help" the enqueuer).

In more detail, an enqueuer creates a new node with the new value to be enqueued, reads `tail`, and finds the node that appears to be last. To verify that node is indeed last, it checks whether the node has a successor. If the node does not have a successor, the thread attempts to append the new node using a CAS. If the CAS succeeds, the thread uses a second CAS to advance `tail` to the new node. Even if this second CAS fails, the thread can still return successfully because, as it happens, the CAS fails only if some other thread "helped" it by advancing `tail`. If the tail node has a successor, then the method tries to "help" other threads by advancing `tail` to refer directly to the successor before trying again to insert its own node.

The `dequeue` method checks that the queue is nonempty by checking that the next field of the head node is not null. It then executes a CAS to change `head` from the sentinel node to its successor, making the successor the new sentinel node. There is, however, a subtle issue: before advancing `head` one must make sure that `tail` is not left referring to the sentinel node which is about to be removed from the queue. To avoid this problem `dequeue` performs a test: if `head` equals `tail` and the (sentinel) node they refer to has a non-null `next` field, then the tail is deemed to be lagging behind. As in the `enqueue` method, `dequeue` then attempts to help make `tail` consistent by swinging it to the sentinel node's successor, and only then updates `head` to remove the sentinel. The value is read from the successor of the sentinel node.

Finally, `isEmpty` is just a simplified version of `dequeue`.

Formal setting. We define an extension RAM_μ of RAM with the kinds of permissions we need to verify the example. Let the set of permissions be:

$$\mathsf{Perm}_\mu = \{1, \mu\} \times \{a, i\}.$$

The permission $(1, a)$ denotes the full permission for a memory cell, and it allows reading from and writing to a memory cell (recall that disposing a cell is forbiden in our language as discussed in Section 2). The permission (μ, a) allows reading a cell; however, unlike read permissions in RAM_p it does not prohibit other threads from writing to the cell. Hence, the value read using a μ-permission can be arbitrary. Permissions $(1, i)$ and (μ, i) have the same meaning, except that additionally the number stored in the cell can only be increased; thus, it is not possible to decrement the contents of the cell using such a permission, and for two successive reads of u_1 and u_2 from the cell using a (μ, i) permission, we always have $u_1 \leq u_2$. A part of our intuition behind Perm_μ is formalised by the following partial operation $\cdot$ on Perm_μ:

$$(\mu, m) \cdot (\mu, m') = \begin{cases} (\mu, m), & \text{if } m = m'; \\ \text{undefined}, & \text{otherwise.} \end{cases}$$

$$(\mu, m) \cdot (1, m') = (1, m') \cdot (\mu, m) = \begin{cases} (1, m), & \text{if } m = m'; \\ \text{undefined}, & \text{otherwise.} \end{cases}$$

$$(1, m) \cdot (1, m') = \text{undefined}.$$

Note that any number of (μ, m)-permissions can be generated by a full permission.

Using Perm_μ, we now define RAM_μ as follows:

$$\mathsf{Loc} = \mathbb{N}^+ \qquad \mathsf{Val} = \mathbb{Z}$$
$$\mathsf{RAM}_\mu = \mathsf{Loc} \rightarrow_{fin} (\mathcal{P}(\mathsf{Val}) \times \mathsf{Perm}_\mu).$$

For $A \subseteq \mathsf{Val}$, we write $\mathsf{upclosed}(A)$ to mean that

$$\forall v, v' \in \mathsf{Val}.\ v \leq v' \wedge v \in A \Rightarrow v' \in A.$$

Figure 9. Transition relation for sample primitive commands in the RAM_μ model. The evaluation of expressions $[\![E]\!]$ ignores the permission part of the model, and it is nondeterministic. It returns an element in $\mathcal{P}(\mathsf{Val} \cup \{\top\})$.

skip, θ	$\rightsquigarrow_t \theta$	
$[E]{=}E', \theta$	$\rightsquigarrow_t \theta[v : (\{v'\}, \pi)]$	if $[\![E]\!]_{\theta,t} \subseteq \mathsf{dom}(\theta)$, $[\![E']\!]_{\theta,t} \subseteq \mathsf{Val}$, $v \in [\![E]\!]_{\theta,t}$, $v' \in [\![E']\!]_{\theta,t}$, $\theta(v) = (\_, (1, a))$
$[E]{=}E', \theta$	$\rightsquigarrow_t \theta[v : (\{v'\}, \pi)]$	if $[\![E]\!]_{\theta,t} \subseteq \mathsf{dom}(\theta)$, $[\![E']\!]_{\theta,t} \subseteq \mathsf{Val}$, $v \in [\![E]\!]_{\theta,t}$, $v' \in [\![E']\!]_{\theta,t}$, $\theta(v) = (V, (1, i))$, $\forall x \in V.\ v' \geq x$
$[E]{=}E', \theta$	$\rightsquigarrow_t \top$	if the above condition does not hold
$\mathsf{assume}(E), \theta$	$\rightsquigarrow_t \theta$	if $[\![E]\!]_{\theta,t} \subseteq \mathsf{Val}$ and $\exists v.\ v \in [\![E]\!]_{\theta,t} \wedge v \neq 0$
$\mathsf{assume}(E), \theta$	$\rightsquigarrow_t \top$	if $[\![E]\!]_{\theta,t} \not\subseteq \mathsf{Val}$

The $*$ operator on RAM_μ is defined as follows: for all $x \in \mathsf{Loc}$

$$(\theta_0 * \theta_1)(x) = \begin{cases} \theta_0(x), & \text{if } \theta_1(x)\uparrow; \\ \theta_1(x), & \text{if } \theta_0(x)\uparrow; \\ (v_0 \cap v_1, \pi_0 \cdot \pi_1), & \text{if } \theta_i(x) = (v_i, \pi_i),\ \pi_i = (\mu, \_); \\ (v_i, \pi_0 \cdot \pi_1), & \text{if } \theta_i(x) = (v_i, \pi_i),\ \pi_i = (1, a),\ v_{|i-1|} = \mathsf{Val}; \\ (v_i, \pi_0 \cdot \pi_1), & \text{if } \theta_i(x) = (v_i, \pi_i),\ \pi_i = (1, i),\ v_i \subseteq v_{|i-1|},\ \mathsf{upclosed}(v_{|i-1|}); \\ \text{undefined}, & \text{otherwise.} \end{cases}$$

The definition of the semantics of the primitive commands is adjusted straightforwardly from the one over the domain RAM_p of Section A; see Figure 9.

The ad-hoc yet flexible permissions supported by RAM_μ are designed mainly to handle the state sharing between the queue and the memory allocator. In more advanced concurrency logics [8], these ad hoc permissions can be defined inside the logic, rather than being hardcoded into the model.

An interesting feature of RAM_μ is that the $*$ operation on it is not cancellative. For example:

$$[1 : (\{1\}, (1, a))] * [2 : (\{1\}, (1, a))] = [1 : (\{1\}, (1, a))] * ([2 : (\{1\}, (1, a))] * [1 : (\{1\}, (\mu, a))]).$$

The issue is that we might either leave μ-snapshots for memory cells we are cancelling out in the residue or not. Fortunately, our requirements on models of program states can be relaxed slightly as follows. The cancellativity of $*$ is used to define a unique operation for substracting a part of a state satisfying a precise predicate that we use while defining the semantics of open programs in Figure 2. There, given a state θ and a precise predicate p describing a pre- or postcondition, we compute a state θ' such that $\theta = \theta' * \theta_p$ for some $\theta_p \in p$. Let us denote the resulting θ' with $\theta \setminus p$, which is defined uniquely when $*$ is cancellative. In our proofs, the predicates p we use in this context describe only full permissions. We can define the result of $\theta \setminus p$ for such predicates p as the state θ' such that $\theta = \theta' * \theta_p$, $\theta_p \in p$ and

$$\forall x.\ (\theta_p(x)\downarrow \Rightarrow \theta'(x)\downarrow) \wedge (\forall V.\ \theta_p(x) = (V, (1, i)) \Rightarrow \theta'(x) = (V, (\mu, i))).$$

It is easy to check that the latter condition makes θ' defined uniquely, by mandating that we leave μ-snapshots for all the memory cells we are transferring in θ_p.

A similar issue arises with the defintion of the δ function. Because $*$ is not cancellative, Property 2 in Definition 2 is not satisfied. We use this property to define the $\setminus$ operation on footprints, substracting the footprint of a state satisfying a pre- or postcondition. As in the case of $*$, we can define the result of $l_1 \setminus l_2$ directly when $l_2 = \delta(\theta_p)$ for θ_p describing only full permissions. Take $\theta \in l_1$. Then $l_1 \setminus l_2 = \delta(\theta')$, where θ' satisfies the above conditions in the definition of $\setminus$ on states. It is again easy to check that the $\setminus$ operation is well-defined.

With the new definitions of the $\setminus$ operations on states and footprints, the proof of the Abstraction Theorem goes through as be-

fore.

To denote elements of RAM_μ, we extend the assertion language of Appendix D as follows:

$$p ::= \ldots \mid E \mapsto F \mid E \mapsto_\mu \cdot \mid E \mapsto^i F \mid E \mapsto^i_\mu F \mid \mathsf{true}_\mu$$

for a standard expression grammar for E, F, except we do not allow the heap dereference operator $[E']$ to be used inside E and F. The consequence of this condition is that $[\![E]\!]_{\theta,t}$ is always a singleton set of values in Val, and it is independent of θ. From now on, when we write semantics of expressions inside assertions, In the following we thus write $[\![E]\!]_t$ instead of $[\![E]\!]_{\theta,t}$ and treat it as an element of Val, instead of a singleton set.

The meaning of assertions is as follows:

- $E \mapsto F$ denotes heaps with a full permission for a single cell at the address E storing F;
- $E \mapsto_\mu \cdot$ denotes a singleton heap with cell E having the permission (μ, a) and storing the set of all values Val;
- $E \mapsto^i_\mu F$ denotes a heap with cell E having the permission $(1, i)$ and storing the set of all values greater than F;
- $E \mapsto^i F$ denotes a heap with cell E having the permission $(1, i)$ and storing the set of all values greater than F;
- true_μ denotes a heap with μ-permissions only.

Formally:

$$\begin{array}{lcl}
\theta, \mathbf{i} \models E \mapsto F & \iff & \theta = [[\![E]\!]_t : (\{[\![F]\!]_t\}, (1, a))] \\
\theta, \mathbf{i} \models E \mapsto_\mu \cdot & \iff & \theta = [[\![E]\!]_t : \mathsf{Val}, (\mu, a))] \\
\theta, \mathbf{i} \models E \mapsto^i F & \iff & \theta = [[\![E]\!]_t : (\{[\![F]\!]_t\}, (1, i))] \\
\theta, \mathbf{i} \models E \mapsto^i_\mu F & \iff & \theta = [[\![E]\!]_t : (\{v \mid v \geq [\![F]\!]_t\}, (\mu, i))] \\
\theta, \mathbf{i} \models \mathsf{true}_\mu & \iff & \forall x.\, \theta(x)\downarrow \Rightarrow \theta(x) = (_, (\mu, _))
\end{array}$$

All assertions used in the proof are implicitly conjoined with $* \, \mathsf{true}_\mu$. For a field `f` of a C structure, we use $E.\mathtt{f} \mapsto E'$ as a shortcut for $E + \mathit{off} \mapsto E'$, where *off* is the offset of `f` in the structure. We also use this notation for the other kinds of $\mapsto$ predicates.

Recall that our programming language (Section 2) does not allow procedures to have local variables, parameters or return values, but provides atomic commands depending on the current thread identifier. In the following we use global arrays indexed by thread identifiers to represent such variables. This is enough, since we do not allow recursive procedures. In particular, the return value is always represented by the array `ret`, and a procedure parameter by an array with an identical name. For a global array `var`, we write $\mathit{var} \Vdash P$ for $\exists \mathit{var}.\, \mathtt{var}[\mathsf{tid}] \mapsto \mathit{var} * P$ and $\lfloor \mathit{var} \rfloor \Vdash P$ for $\exists \mathit{var}.\, \lfloor \mathtt{var}[\mathsf{tid}] \mapsto \mathit{var} \rfloor * P$. Note that here `var` is a physical array, whereas var is a logical variable representing a value of one of its slots. We assume that global variables are allocated at fixed addresses.

We extend the setting of of Section D in two ways, which can be easily accommodated. First, we allow an invocation of a library method to receive the ownership of fixed thread-local data structures representing its local variables, in addition to the shared invariant $\boxed{\mathsf{inv}}$.

Second, while verifying the code of the abstract implementation inside $(\!|\cdot|\!)$, we are allowed to treat non-determinism angelically, choosing the values returned by non-deterministic expressions ourselves. This is because the proof of linearizability *constructs* the execution of the abstract implementation linearizing a given execution of the concrete one. To incorporate such non-determinism, we allow the abstract library implementation to use a special `havoc(x)` command, which assigns a non-deterministic value to the address x and is treated angelically using the following proof rule:

$$\frac{}{\{E \mapsto \_\}\, \mathtt{havoc}(E)\, \{E \mapsto F\}}\ \textsc{Havoc}$$

The rule allows choosing any expression F as the value written to E.

Finally, our proofs use prophecy variables[4], which are often needed in proofs of linearizability [32].

In the following proofs we use $\mathtt{CAS}_{\mathsf{A},\mathsf{B}}(\mathtt{addr}, \mathtt{v1}, \mathtt{v2})$ as a shortcut for

```
if (nondet()) {
  atomic_A {
    assume(*addr == v1); *addr = v2;
  }
  return 1;
} else {
  atomic_B {
    assume(*addr != v1);
  }
  return 0;
}
```

This definition is semantically equivalent to the definition given at the beginning of this section, but allows different action annotations for the successful and the failure cases. We also use $\mathtt{CAS}_{\mathtt{f},\mathsf{A},\mathsf{B}}(\mathtt{addr}, \mathtt{v1}, \mathtt{v2})$ as a shortcut for

```
if (nondet()) {
  atomic_A {
    assume(*addr == v1); *addr = v2; (|f|);
  }
  return 1;
} else {
  atomic_B {
    assume(*addr != v1);
  }
  return 0;
}
```

Specifications. The methods of the two libraries satisfy the following specifications:

$\{\mathit{ret} \Vdash \mathsf{emp}\}\ \mathtt{alloc}()$
$\quad \{\mathit{ret} \Vdash (\mathit{ret} = 0 \wedge \mathsf{emp}) \vee (\mathit{ret} \neq 0 \wedge \mathsf{Block}(\mathit{ret}))\}$

$\{\mathit{block} \Vdash \mathsf{Block}(\mathit{block})\}\ \mathtt{free}(\mathtt{block})\ \{\mathit{block} \Vdash \mathsf{emp}\}$

$\{\mathit{val}, \mathit{ret} \Vdash \mathsf{emp}\}\ \mathtt{enqueue}(\mathtt{val})$
$\quad \{\mathit{val}, \mathit{ret} \Vdash (\mathit{ret} = \mathtt{SUCCESS} \vee \mathit{ret} = \mathtt{FAIL}) \wedge \mathsf{emp}\}$

$\{\mathit{ret} \Vdash \mathsf{emp}\}\ \mathtt{dequeue}()\ \{\mathit{ret} \Vdash \mathsf{emp}\}$

$\{\mathit{ret} \Vdash \mathsf{emp}\}\ \mathtt{isEmpty}()\ \{\mathit{ret} \Vdash (\mathit{ret} = 0 \vee \mathit{ret} = 1) \wedge \mathsf{emp}\}$

where

$$\mathsf{Block}(x) = x.\mathtt{next.ptr} \mapsto \_ * x.\mathtt{next.count} \mapsto^i \_ * x.\mathtt{val} \mapsto \_$$

The Block predicate describes permissions we associate with `Node` structures while verifying the queue algorithm. The abstract atomic implementations of the libraries are given in Figures 10 and 11. These use abstract cells storing a sequence of queue elements or a set of pointers to free memory blocks in the allocator. We assume sequential primitive operations on such sequences and sets. Note that, even though the abstract implementation of `enqueue` does not call the memory allocator, it can still fail non-deterministically, to simulate the behaviour of the concrete implementation. For the same reason, `alloc` non-deterministically changes the contents of the first $\mathtt{sizeof}(\mathtt{Block}*)$ bytes of the block it allocates.

Linearizability of the allocator. The detailed proofs of linearizability are given in Section E.1 below. The linearization point of a `free` operation is at the successful CAS. If an `alloc` operation returns `NULL`, its linearization point is at at the point when it reads `NULL` from `Top`. Otherwise, it is at the successful CAS. It is easy to check that all assertions used in the proof are stable. We also show the treatment of the CAS operations using AtomicOut and LinPoint rules in detail. For readability we inlined the bodies of abstract methods called at linearization points, so that `return` statements in them are meant to jump to the closing bracket of $(\!|\cdot|\!)$.

[4] M. Abadi and L. Lamport. The existence of refinement mappings. *TCS*, 1991

Figure 10. Abstract queue implementation

```
abstract Sequence<int> Queue;

int atomic enqueue_abs(int val) {
  if (havoc()) {
    add_to_tail(Queue, val)
    return SUCCESS;
  } else {
    return FAIL;
  }
}

int atomic dequeue_abs() {
  if (!isEmpty(Queue))
    return remove_head(Queue);
  else
    return EMPTY;
}
```

Figure 11. Abstract memory allocator implementation

```
struct Block { Block *next; };
abstract Set<void*> Mem;

void atomic free_abs(void *block) {
  add(Mem, block);
}

void* atomic alloc_abs() {
  void *b;
  if (!isEmpty(Mem)) {
    b = remove(Mem);
    havoc(b, sizeof(Block*));
    return b;
  } else {
    return NULL;
  }
}
```

Linearizability of the queue. As we argued in Section 1, to verify that the queue is linearizable, we need to refer to the state of the memory allocator. This would complicate the proof if we considered the original allocator implementation. The Abstraction Theorem allows us to avoid this using the method of abstracting nested libraries in Section 5.1: as we explained in Section 6, we first replace the memory allocator with its atomic implementation and then prove the linearizability of the queue using it.

The proof is analogous to existing proofs in the literature[5]; the only added feature is the treatment of interactions with the memory allocator. The linearization points for queue methods are as follows. A successful `enqueue` is linearized at the point when the thread executes a CAS to link the new node into the list representing the queue. If `dequeue` returns a value, then its linearization point occurs when it completes a successful CAS to change `head`; otherwise it is linearized at the moment when it reads `first.ptr->next` for the last time before returning.

We note that the queue might also function as a container, storing data structures of a certain type instead of integers. The proof of its linearizability can be generalised to this case on the lines of the above proof of the memory allocator.

E.1 Detailed linearizability proofs

Auxiliary definitions for the proof of the linearizability of the memory allocator.

$$\mathsf{Block}(x,y) = x.\mathtt{next.ptr} \mapsto y * x.\mathtt{next.count} \mapsto^{i} \_ * x.\mathtt{val} \mapsto \_$$

$$\mathsf{Block}(x) = \exists y.\, \mathsf{Block}(x,y)$$

$$\mathsf{PBlock}(x) = x.\mathtt{next.count} \mapsto^{i} \_ * x.\mathtt{val} \mapsto \_$$

$$\mathsf{RBlock}(x,y) = \exists X.\, x \mapsto y * X * \lfloor x \mapsto \_ * X \rfloor \wedge (\mathsf{PBlock}(x))(X)$$

$$\&\mathtt{Top.ptr} \mapsto y * \&\mathtt{Top.count} \mapsto c * \lfloor \mathsf{Mem} \mapsto M \rfloor \rightsquigarrow \&\mathtt{Top.ptr} \mapsto x * \&\mathtt{Top.count} \mapsto (c{+}1) * \mathsf{RBlock}(x,y) * \lfloor (\mathsf{Mem} \mapsto M \uplus \{x\}) \rfloor \quad (\mathsf{Push})$$

$$\&\mathtt{Top.ptr} \mapsto x * \&\mathtt{Top.count} \mapsto c * \mathsf{RBlock}(x,y) * \lfloor (\mathsf{Mem} \mapsto M \uplus \{x\}) \rfloor \rightsquigarrow \&\mathtt{Top.ptr} \mapsto y * \&\mathtt{Top.count} \mapsto (c{+}1) * \lfloor \mathsf{Mem} \mapsto M \rfloor \quad (\mathsf{Pop})$$

$$\mathsf{emp} \rightsquigarrow \mathsf{emp} \quad (\mathsf{Id})$$

$$G = \mathsf{Push} \cup \mathsf{Pop} \cup \mathsf{Id}$$

$$\mathsf{ls}(x,y,M) \Leftrightarrow (x = y \wedge M = \emptyset \wedge \mathsf{emp}) \vee (x \neq y \wedge x \in M \wedge \exists z.\, \mathsf{RBlock}(x,z) * \mathsf{ls}(z,y,M - \{x\}))$$

$$\mathsf{StackInv}(c) = \exists M, x.\, \&\mathtt{Top.ptr} \mapsto x * \&\mathtt{Top.count} \mapsto c * \mathsf{ls}(x,0,M) * \lfloor \mathsf{Mem} \mapsto M \rfloor$$

$$\mathsf{StackInv} = \exists c.\, \mathsf{StackInv}(c)$$

$$S(x,c,y) = \exists M.\, \&\mathtt{Top.ptr} \mapsto x * \&\mathtt{Top.count} \mapsto c * \mathsf{RBlock}(x,y) * \mathsf{ls}(y,0,M) * \lfloor \mathsf{Mem} \mapsto M \rfloor$$

Proof of the linearizability of the memory allocator.

```
struct Block { Block *next; };
struct BlockRef { Block *ptr; unsigned count; };
BlockRef Top;
abstract Set<void*> Mem;

void free(void *block) {
  BlockRef t, x;
  {t, x ⊩ ∃X. Pre(X) ∧ X * StackInv ∧ (block ⊩ Block(block))(X)}
  do {
    {t, x, block ⊩ ∃X. Pre(block[tid] ↦ block * X) ∧
     X * StackInv}
    atomic_Id { t = Top; }
    (Block*)block->next = t.ptr;
    x = BlockRef((Block*)block, t.count+1);
    {t, x, block ⊩ Pre(block[tid] ↦ block * block ↦ _ * X) ∧
     block ↦ t.ptr * X * StackInv ∧ (PBlock(block))(X) ∧
     x = BlockRef(block, t.count+1)}
  } while (!CAS_{free_abs(block),Push,Id}(&Top, t, x));
  {t, x, block ⊩ Post(block[tid] ↦ block) ∧ StackInv}
}
{t, x, block ⊩ Post(block[tid] ↦ block) ∧ StackInv}

void *alloc() {
  BlockRef t, x;
  {t, x, ret, ⌊b⌋ ⊩ Pre(ret[tid] ↦ ret) ∧ StackInv}
  do {
    {t, x, ret, ⌊b⌋ ⊩ Pre(ret[tid] ↦ ret) ∧ StackInv}
    atomic_Id { t = Top; if (t.ptr == NULL) alloc_abs(); }
    {t, x, ret, ⌊b⌋ ⊩ (t.ptr = 0 ∧ Post(ret[tid] ↦ 0) ∧ StackInv) ∨
     (Pre(ret[tid] ↦ ret) ∧
     ((∃c. StackInv(c) * t.ptr.next ↦_μ · ∧ c > t.count) ∨
     (∃y. S(t.ptr, t.count, y) * t.ptr.next ↦_μ ·)))}
    if (t.ptr == NULL)
      {t, x, ret, ⌊b⌋ ⊩ Post(ret[tid] ↦ 0) ∧ StackInv}
      return NULL;
    {t, x, ret, ⌊b⌋ ⊩ Pre(ret[tid] ↦ ret) ∧
     ((∃c. StackInv(c) * t.ptr.next ↦_μ · ∧ c > t.count) ∨
     (∃y. S(t.ptr, t.count, y) * t.ptr.next ↦_μ ·))}
    atomic_Id { x = BlockRef(t.ptr->next, t.count+1); }
    {t, x, ret, ⌊b⌋ ⊩ Pre(ret[tid] ↦ ret) ∧
     ((∃c. StackInv(c) ∧ c > t.count) ∨
     (∃y. S(t.ptr, t.count, y) ∧ x = BlockRef(y, t.count+1)))}
  } while (!CAS_{alloc_abs(),Pop,Id}(&Top, t, x));
  {t, x, ret, ⌊b⌋ ⊩ ∃X. Post(ret[tid] ↦ t.ptr * X) ∧ X * StackInv ∧
   (Block(t.ptr))(X)}
  return t.ptr;
```

[5] V. Vafeiadis. Shape-value abstraction for verifying linearizability. In *VMCAI*, 2009.

```
}
{t, x, ret, ⌊b⌋ ⊩ StackInv ∗ (ret = 0 ∧ Post(ret[tid] ↦ 0)) ∨
 (∃X. Post(ret[tid] ↦ ret ∗ X) ∧ X ∧ (Block(ret))(X))}
```

Linearization point of `free` in detail.

$\mathtt{CAS}_{\mathtt{free\_abs(block)},\mathsf{Push},\mathsf{Id}}$(&Top, t, x):

```
{t, x, block ⊩ ∃X. Pre(block[tid] ↦ block ∗ block ↦ _ ∗ X) ∧
 block ↦ t.ptr ∗ X ∗ StackInv ∧ (PBlock(block))(X) ∧
 x = BlockRef(block, t.count+1)}
if (nondet()) {
  atomic_Push {
    assume(Top == t);
    {t, x, block ⊩ ∃X. Pre(block[tid] ↦ block ∗ block ↦ _ ∗ X) ∧
     block ↦ t.ptr ∗ X ∗ &Top.ptr ↦ t.ptr ∗
     &Top.count ↦ t.count ∗ ⌊Mem ↦ M⌋ ∧
     x = BlockRef(block, t.count+1) ∧ (PBlock(block))(X)}
    Top = x;
    {t, x, block ⊩ ∃X. Pre(block[tid] ↦ block ∗ block ↦ _ ∗ X)
     &Top.ptr ↦ block ∗ &Top.count ↦ (t.count+1) ∗
     block ↦ t.ptr ∗ X ∗ ⌊Mem ↦ M⌋ ∧ (PBlock(block))(X)}
    ⟪
      {t, x, block ⊩ ⌊Mem ↦ M ∗ block[tid] ↦ block⌋ ∗
       &Top.ptr ↦ block ∗ &Top.count ↦ (t.count+1) ∗
       RBlock(block, t.ptr)}
      add(Mem, block);
      {t, x, block ⊩ ⌊(Mem ↦ M⊎{block})∗block[tid] ↦ block⌋∗
       &Top.ptr ↦ block ∗ &Top.count ↦ (t.count+1) ∗
       RBlock(block, t.ptr)}
    ⟫
    {t, x, block ⊩ Post(block[tid] ↦ block) ∧
     &Top.ptr ↦ block ∗ &Top.count ↦ (t.count+1) ∗
     ⌊Mem ↦ M ⊎ {block}⌋ ∗ RBlock(block, t.ptr)}
  }
  {t, x, block ⊩ Post(block[tid] ↦ block) ∧ StackInv}
  return 1;
} else {
  atomic_Id {
    assume(Top != t);
  }
  {t, x, block ⊩ ∃X. Pre(block[tid] ↦ block ∗ block ↦ _ ∗ X) ∧
   block ↦ t.ptr ∗ X ∗ StackInv}
  return 0;
}
```

Linearization point of `alloc` in detail.

$\mathtt{CAS}_{\mathtt{alloc\_abs()},\mathsf{Pop},\mathsf{Id}}$(&Top, t, x):

```
{t, x, ret, ⌊b⌋ ⊩ Pre(ret[tid] ↦ ret) ∧
 ((∃c. StackInv(c) ∧ c > t.count) ∨
 (∃y. S(t.ptr, t.count, y) ∧ x = BlockRef(y, t.count+1)))}
if (nondet()) {
  atomic_Pop {
    assume(Top == t);
    {t, x, ret, ⌊b⌋ ⊩ Pre(ret[tid] ↦ ret) ∗ RBlock(t.ptr, y) ∗
     &Top.ptr ↦ t.ptr ∗ &Top.count ↦ t.count ∗
     ⌊(Mem ↦ M ⊎ {t.ptr})⌋ ∧
     x = BlockRef(y, t.count+1)}
    Top = x;
    {t, x, ret, ⌊b⌋ ⊩ Pre(ret[tid] ↦ ret) ∗ RBlock(t.ptr, y) ∗
     &Top.ptr ↦ y ∗ &Top.count ↦ (t.count+1) ∗
     ⌊(Mem ↦ M ⊎ {t.ptr})⌋}
    ⟪
      void *b;
      {t, x, ret ⊩ ⌊b ⊩ ret[tid] ↦ _⌋ ∗ RBlock(t.ptr, y) ∗
       &Top.ptr ↦ y ∗ &Top.count ↦ (t.count+1) ∗
       ⌊(Mem ↦ M ⊎ {t.ptr})⌋}
      if (!isEmpty(Mem)) {
        b = remove(Mem);
        {t, x, ret ⊩ ⌊b ⊩ ret[tid] ↦ _ ∧ b = t.ptr⌋ ∗
         RBlock(t.ptr, y) ∗ ⌊Mem ↦ M⌋ ∗
         &Top.ptr ↦ y ∗ &Top.count ↦ (t.count+1)}
        havoc(b, sizeof(Block*));
      } else {
        {false}
        b = NULL;
      }
      {t, x, ret ⊩ ∃X. ⌊b ⊩ ret[tid] ↦ _ ∗ X ∧ b = t.ptr⌋ ∗ X ∗
       &Top.ptr ↦ y ∗ &Top.count ↦ (t.count+1) ∧
       (Block(t.ptr))(X)}
      return block;
      {t, x, ret ⊩ ∃X. ⌊b ⊩ ret[tid] ↦ t.ptr ∗ X⌋ ∗ X ∗
       &Top.ptr ↦ y ∗ &Top.count ↦ (t.count+1) ∧
       (Block(t.ptr))(X)}
    ⟫
    {t, x, ret, ⌊b⌋ ⊩ ∃X. Post(ret[tid] ↦ t.ptr ∗ X) ∧ X ∗
     &Top.ptr ↦ y ∗ &Top.count ↦ (t.count+1) ∧
     (Block(t.ptr))(X)}
  }
  return 1;
} else {
  atomic_Pop {
    assume(Top != t);
  }
  {t, x, ret ⊩ Pre(ret[tid] ↦ ret) ∧ StackInv}
  return 0;
}
```

Auxiliary definitions for the proof of the non-blocking queue.

$$\mathsf{Block}(x, y, z) = x.\mathtt{next.ptr} \mapsto y * x.\mathtt{next.count} \mapsto^i \_ * x.\mathtt{val} \mapsto z$$

$$\mathsf{Block}(x, y) = \mathsf{Block}(x, y, \_)$$

$$\mathsf{Block}(x) = \mathsf{Block}(x, \_, \_)$$

$$\mathsf{RBlock}(x) = \exists X.\, X * \lfloor X \rfloor \wedge (\mathsf{Block}(x))(X)$$

$$(\mathsf{Mem} \mapsto M_c \uplus \{x\}) * \mathsf{Block}(x) \rightsquigarrow \mathsf{Mem} \mapsto M_c \qquad \text{(Alloc)}$$

$$\mathsf{Mem} \mapsto M_c \rightsquigarrow (\mathsf{Mem} \mapsto M_c \cup \{x\}) * \mathsf{Block}(x) \qquad \text{(Free)}$$

$$\begin{aligned}&\&\mathtt{tail.ptr} \mapsto x * \&\mathtt{tail.count} \mapsto^i c' *\\ &x.\mathtt{ptr} \mapsto 0 * x.\mathtt{count} \mapsto^i c * x.\mathtt{val} \mapsto a * \mathsf{Queue} \mapsto \alpha a \rightsquigarrow\\ &\quad \&\mathtt{tail.ptr} \mapsto x * \&\mathtt{tail.count} \mapsto^i c' *\\ &\quad x.\mathtt{ptr} \mapsto y * x.\mathtt{count} \mapsto^i (c{+}1) * x.\mathtt{val} \mapsto a *\\ &\quad y.\mathtt{ptr} \mapsto 0 * y.\mathtt{count} \mapsto^i \_ * y.\mathtt{val} \mapsto b * \mathsf{Queue} \mapsto \alpha a b\end{aligned} \qquad \text{(Insert)}$$

$$\begin{aligned}&t \neq x \wedge \&\mathtt{head.ptr} \mapsto x * \&\mathtt{head.count} \mapsto^i c * \&\mathtt{tail.ptr} \mapsto t *\\ &x.\mathtt{ptr} \mapsto y * x.\mathtt{count} \mapsto^i c_1 * x.\mathtt{val} \mapsto \_\\ &y.\mathtt{ptr} \mapsto z * y.\mathtt{count} \mapsto^i c_2 * y.\mathtt{val} \mapsto a * \mathsf{Queue} \mapsto a\alpha \rightsquigarrow\\ &\quad \&\mathtt{head.ptr} \mapsto y * \&\mathtt{head.count} \mapsto^i (c{+}1) *\\ &\quad y.\mathtt{ptr} \mapsto z * y.\mathtt{count} \mapsto^i c_2 * y.\mathtt{val} \mapsto \_ * \mathsf{Queue} \mapsto \alpha\end{aligned} \qquad \text{(Remove)}$$

$$\begin{aligned}&\&\mathtt{tail.ptr} \mapsto x * \&\mathtt{tail.count} \mapsto^i c *\\ &x.\mathtt{ptr} \mapsto y * x.\mathtt{count} \mapsto^i c_1 * y.\mathtt{ptr} \mapsto 0 * y.\mathtt{count} \mapsto^i c_2 \rightsquigarrow\\ &\quad \&\mathtt{tail.ptr} \mapsto y * \&\mathtt{tail.count} \mapsto^i (c{+}1) *\\ &\quad x.\mathtt{ptr} \mapsto y * x.\mathtt{count} \mapsto^i c_1 * y.\mathtt{ptr} \mapsto 0 * y.\mathtt{count} \mapsto^i c_2\end{aligned} \qquad \text{(Advance)}$$

$$\mathsf{emp} \rightsquigarrow \mathsf{emp} \qquad \text{(Id)}$$

$$G = \mathsf{Alloc} \cup \mathsf{Free} \cup \mathsf{Insert} \cup \mathsf{Remove} \cup \mathsf{Advance} \cup \mathsf{Id}$$

$$\begin{aligned}&\mathsf{ls}(h, t, \alpha, M_a, x_c) \Leftrightarrow (h = t \wedge ((\exists v.\, \alpha = v \wedge x_p = 0 \wedge t \in M_a \wedge\\ &t.\mathtt{ptr} \mapsto 0 * t.\mathtt{count} \mapsto^i x_c * t.\mathtt{val} \mapsto v) \vee (\exists v_1, v_2.\, \alpha = v_1 v_2 \wedge\\ &x_p = 0 \wedge x_p, t \in M_a \wedge t.\mathtt{ptr} \mapsto x_p * t.\mathtt{count} \mapsto^i x_c * t.\mathtt{val} \mapsto v_1 *\\ &\quad x_p.\mathtt{ptr} \mapsto 0 * x_p.\mathtt{count} \mapsto^i \_ * x_p.\mathtt{val} \mapsto v_2))) \vee\\ &(h \neq t \wedge \exists \alpha', z, v.\, \alpha = v\alpha' \wedge h \in M_a \wedge \mathsf{Block}(h, z, v) * \mathsf{ls}(z, t, \alpha'))\end{aligned}$$

$$
\begin{array}{c}
Q(h_p, h_c, n_t, n_c, t_p, t_c, x_p, x_c, M_a) = \\
\exists M_c, \alpha.\ M_c \subseteq M_a \wedge \mathsf{Mem} \mapsto M_c * \\
\lfloor \mathsf{Mem} \mapsto M_a * (\circledast_{u \in M_a - M_c} \mathsf{Block}(u)) * \mathsf{Queue} \mapsto \alpha \rfloor * \\
(\circledast_{u \in M_c} \mathsf{RBlock}(u)) * \&\mathtt{head.ptr} \mapsto h_p * \&\mathtt{head.count} \mapsto^i h_c * \\
\&\mathtt{tail.ptr} \mapsto t_p * \&\mathtt{tail.count} \mapsto^i t_c \wedge \\
(h_p = t_p \wedge \alpha = \varepsilon \wedge n_p = x_p \wedge n_c = x_c \wedge t_p \in M_a \wedge \\
t_p.\mathtt{ptr} \mapsto 0 * t_p.\mathtt{count} \mapsto^i x_c * t_p.\mathtt{val} \mapsto \_) \vee \\
(h_p = t_p \wedge \alpha = v \wedge n_p = x_p \wedge n_c = x_c \wedge t_p, x_p \in M_a \wedge \\
t_p.\mathtt{ptr} \mapsto x_p * t_p.\mathtt{count} \mapsto^i x_c * t_p.\mathtt{val} \mapsto \_ * \\
x_p.\mathtt{ptr} \mapsto 0 * x_p.\mathtt{count} \mapsto^i \_ * x_p.\mathtt{val} \mapsto v) \vee \\
(h_p \neq t_p \wedge h_p.\mathtt{ptr} \mapsto n_p * h_p.\mathtt{count} \mapsto^i n_c * h_p.\mathtt{val} \mapsto \_ * \\
\mathsf{ls}(n_p, t_p, \alpha, M_a))
\end{array}
$$

$$
Q_1(t_p, t_c, x_p, x_c, M_a) = Q(\_, \_, \_, \_, t_p, t_c, x_p, x_c, M_a)
$$

$$
Q_2(h_p, h_c, n_p, n_c, t_p, t_c, M_a) = Q(h_p, h_c, n_p, n_c, t_p, t_c, \_, \_, M_a)
$$

$$
\mathsf{QueueInv} = Q(\_, \_, \_, \_, \_, \_, \_, \_, \_)
$$

Proof of the linearizability of enqueue**.**

```
int enqueue(int val) {
  Node *node;
  NodeRef next, last;
  {node, next, last, val, ret ⊩ QueueInv ∧
   Pre(val[tid] ↦ val * ret[tid] ↦ ret)}
  node = alloc_abs_Alloc();
  {node, next, last, val, ret ⊩ Q1(_, _, _, _, {node} ∪ _) ∧
   Pre(val[tid] ↦ val * ret[tid] ↦ ret) * ((node = 0 ∧ emp) ∨
   (node.val ↦ _ * node.next.ptr ↦ _ * node.next.count ↦^i _))}
  if (node == NULL) {
    enqueue_abs(val);
    {node, next, last, val, ret ⊩ Q1(_, _, _, _, {node} ∪ _) ∧
     Post(val[tid] ↦ val * ret[tid] ↦ FAIL)}
    return FAIL;
  }
  node->val = val;
  node->next.ptr = NULL;
  while (true) {
    {node, next, last, val, ret ⊩ Q1(_, _, _, _, {node} ∪ _) ∧
     Pre(val[tid] ↦ val * ret[tid] ↦ ret) * node.val ↦ val *
     node.next.ptr ↦ 0 * node.next.count ↦^i _}
    atomic_Id { last = tail; }
    {node, next, last, val, ret ⊩
     (Q1(last.ptr, last.count, _, _, {node} ∪ _) ∨
     (∃c. Q1(_, c, _, _, {node} ∪ _) ∧ c > last.count)) ∧
     Pre(val[tid] ↦ val * ret[tid] ↦ ret) * node.val ↦ val *
     node.next.ptr ↦ 0 * node.next.count ↦^i _ *
     last.ptr.next.ptr ↦_μ · * last.ptr.next.count ↦^i_μ _}
    atomic_Id { next = last.ptr->next; }
    {node, next, last, val, ret ⊩
     ∃c'. last.ptr.next.ptr ↦_μ · * last.ptr.next.count ↦^i_μ c' *
     (Q1(last.ptr, last.count, next.ptr, next.count, {node} ∪ _) ∨
     (QueueInv ∧ c' > next.count ∧ next.ptr = 0) ∨
     (∃c. Q1(_, c, _, _, {node} ∪ _) ∧ c > last.count)) ∧
     Pre(val[tid] ↦ val * ret[tid] ↦ ret) * node.val ↦ val *
     node.next.ptr ↦ 0 * node.next.count ↦^i _}
    if (atomic_Id { tail != last }) continue;
    if (next.ptr == NULL) {
      {node, next, last, val, ret ⊩ next.ptr = 0 ∧
       ∃c'. last.ptr.next.ptr ↦_μ · * last.ptr.next.count ↦^i_μ c' *
       (Q1(last.ptr, last.count, next.ptr, next.count, {node} ∪ _) ∨
       (∃c. Q1(_, _, _, _, {node} ∪ _) ∧ c' > next.count)) ∧
       Pre(val[tid] ↦ val * ret[tid] ↦ ret) * node.val ↦ val *
       node.next.ptr ↦ 0 * node.next.count ↦^i _}
      if (CAS_Insert,Id(last.ptr->next, next,
              NodeRef(node, next.count+1)))
        break;
    } else
      {node, next, last, val, ret ⊩ next.ptr ≠ 0 ∧
       (Q1(last.ptr, last.count, next.ptr, next.count, {node} ∪ _) ∨
       (∃c. Q1(_, c, _, _, {node} ∪ _) ∧ c > last.count)) ∧
       Pre(val[tid] ↦ val * ret[tid] ↦ ret) * node.val ↦ val *
       node.next.ptr ↦ 0 * node.next.count ↦^i _}
      CAS_Advance,Id(tail, last,
          NodeRef(next.ptr, last.count+1));
  }
  {node, next, last, val, ret ⊩ next.ptr ≠ 0 ∧
   (Q1(last.ptr, last.count, node, next.count+1) ∨
   (∃c. Q1(_, c, _, _) ∧ c > last.count)) ∧
   Post(val[tid] ↦ val * ret[tid] ↦ SUCCESS)}
  CAS_enqueue,Advance,Id(tail, node,
      NodeRef(next.ptr, last.count+1));
  {node, next, last, val, ret ⊩ QueueInv
   Post(val[tid] ↦ val * ret[tid] ↦ SUCCESS)}
  return SUCCESS;
}
```

Proof of the linearizability of dequeue**.**

```
int dequeue() {
  NodeRef next, first, last;
  int val;
  {next, first, last, val, ret ⊩ QueueInv ∧
   Pre(ret[tid] ↦ ret)}
  while (true) {
    {next, first, last, val, ret ⊩ QueueInv ∧
     Pre(ret[tid] ↦ ret)}
    atomic_Id { first = head; }
    atomic_Id { last = tail; }
    {next, first, last, val, ret ⊩ Pre(ret[tid] ↦ ret) ∧
     ∃x1, x2, y1, y2. Q2(x1, x2, y1, y2, z1, _, _, _) ∧
     (x1 = first.ptr ∧ x2 = first.count ∧
     y1 = last.ptr ∧ y2 = last.count) ∨ x2 > first.count ∨
     (x1 = first.ptr ∧ x2 = first.count ∧ y2 > last.count ∧
     z1 ≠ 0) ∧ first.ptr ↦_μ · * first.count ↦^i_μ _}
    atomic_Id {
      next = first.ptr->next;
      if (first.ptr == last.ptr && next.ptr == NULL &&
          prophecy(1: head == first))
        dequeue_abs();
    }
    // Note that if the prophecy is true,
    // then now we also have head = first
    {next, first, last, val, ret ⊩
     ∃x1, x2, y1, y2, z1, z2. Q2(x1, x2, y1, y2, z1, z2, _, _) ∧
     (first.ptr ≠ last.ptr ∧ next.ptr ≠ 0 ⇒ (x2 > first.count ∨
     x2 = first.count ∧ x1 = first.ptr ∧
     y1 = next.ptr ∧ y2 = next.count) ∧
     Pre(ret[tid] ↦ ret) ∧ emp) ∧
     (first.ptr = last.ptr ∧ next.ptr ≠ 0 ⇒
     (x2 > first.count ∨ z2 > last.count ∨
     x2 = first.count ∧ x1 = first.ptr ∧
     y1 = next.ptr ∧ y2 = next.count ∧
     z1 = last.ptr ∧ z2 = last.count) ∧
     Pre(ret[tid] ↦ ret) ∧ emp) ∧
     (first.ptr = last.ptr ∧ next.ptr = 0 ⇒
     (x2 > first.count ∧ ¬prophecy(1: head == first) ∧ emp ∨
     x2 = first.count ∧ x1 = first.ptr ∧
     y1 = next.ptr ∧ y2 = next.count ∧
     z1 = last.ptr ∧ z2 = last.count ∧
     prophecy(1: head == first) ∧ Post(ret[tid] ↦ EMPTY))) *
     next.val ↦_μ ·}
    1: if (atomic_Id { head != first }) continue;
    {next, first, last, val, ret ⊩ Pre(ret[tid] ↦ ret) ∧
     ∃x1, x2, y1, y2, z1, z2. Q2(x1, x2, y1, y2, z1, z2, _, _) ∧
     (first.ptr ≠ last.ptr ∧ next.ptr ≠ 0 ⇒ (x2 > first.count ∨
     x2 = first.count ∧ x1 = first.ptr ∧
     y1 = next.ptr ∧ y2 = next.count)) ∧
     (first.ptr = last.ptr ∧ next.ptr ≠ 0 ⇒ z2 > last.count ∨
     x2 = first.count ∧ x1 = first.ptr ∧
     y1 = next.ptr ∧ y2 = next.count ∧
     z1 > last.ptr ∧ z2 > last.count)) ∧
     (first.ptr = last.ptr ∧ next.ptr = 0 ⇒
     Post(ret[tid] ↦ EMPTY)) * next.val ↦_μ ·}
    if (first.ptr == last.ptr) {
      if (next.ptr == NULL) {
```

```
      {next, first, last, val, ret ⊩
       QueueInv ∧ Post(ret[tid] ↦ EMPTY)}
      return EMPTY;
    }
    {next, first, last, val, ret ⊩ Pre(ret[tid] ↦ ret) ∧
     ∃x1, x2, y1, y2, z1, z2. Q2(x1, x2, y1, y2, z1, z2, _, _) ∧
     first.ptr = last.ptr ∧ next.ptr ≠ 0 ∧ (z2 > last.count ∨
     x2 = first.count ∧ x1 = first.ptr ∧
     y1 = next.ptr ∧ y2 = next.count ∧
     z1 = last.ptr ∧ z2 = last.count) ∧ emp}
    CAS(tail, last, NodeRef(next.ptr, last.count+1));
  } else {
    {next, first, last, val, ret ⊩ Pre(ret[tid] ↦ ret) ∧
     ∃x1, x2, y1, y2, z1, z2. Q2(x1, x2, y1, y2, z1, z2, _, _) ∧
     first.ptr ≠ last.ptr ∧ next.ptr ≠ 0 ∧ (x2 > first.count ∨
     x2 = first.count ∧ x1 = first.ptr ∧
     y1 = next.ptr ∧ y2 = next.count) ∧
     next.val ↦μ ·}
    atomic_ld { val = next->val; }
    {next, first, last, val, ret ⊩ Pre(ret[tid] ↦ ret) ∧
     ∃x1, x2, y1, y2, z1, z2, v. Q2(x1, x2, y1, y2, z1, z2, v, _) ∧
     first.ptr ≠ last.ptr ∧ next.ptr ≠ 0 ∧ (x2 > first.count ∨
     x2 = first.count ∧ x1 = first.ptr ∧
     y1 = next.ptr ∧ y2 = next.count ∧ v = next.val) ∧ emp}
    if (CAS_{dequeue_abs(),Remove,ld}(head, first,
            NodeRef(next.ptr, first.count+1)))
      break;
  }
}
{next, first, last, val, ret ⊩
 Q1(_, _, _, _, _, _, {first} ∪ _) ∧ Post(ret[tid] ↦ val) ∗
 (first.val ↦ _ ∗ first.next.ptr ↦ _ ∗ first.next.count ↦^i _))}
free_abs_Free(first.ptr);
{next, first, last, val, ret ⊩ QueueInv ∧ Post(ret[tid] ↦ val)}
return val;
}
```

Proof of the linearizability of `isEmpty`.

```
int isEmpty() {
  NodeRef next, first, last;
  {next, first, last, ret ⊩ QueueInv ∧
   Pre(ret[tid] ↦ ret)}
  while (true) {
    {next, first, last, ret ⊩ QueueInv ∧
     Pre(ret[tid] ↦ ret)}
    atomic_ld { first = head; }
    atomic_ld {
      last = tail;
      if (first.ptr != last.ptr && prophecy(l: head == first))
        isEmpty_abs();
    }
    // Note that if the prophecy is true,
    // then now we also have head = first
    {next, first, last, ret ⊩
     ∃x1, x2, y1, y2. Q2(x1, x2, y1, y2, z1, _, _, _) ∧
     (x1 = first.ptr ∧ x2 = first.count ∧
     y1 = last.ptr ∧ y2 = last.count) ∨ x2 > first.count ∨
     (x1 = first.ptr ∧ x2 = first.count ∧ y2 > last.count ∧
     z1 ≠ 0) ∧ first.ptr ↦μ · ∗ first.count ↦^i_μ _ ∧
     (first.ptr ≠ last.ptr ∧ prophecy(l: head == first) ∧
     Post(ret[tid] ↦ 0 ∨ (first.ptr = last.ptr ∨
     ¬prophecy(l: head == first)) ∧ Pre(ret[tid] ↦ ret))}
    atomic_ld {
      next = first.ptr->next;
      if (first.ptr == last.ptr && next.ptr == NULL &&
          prophecy(l: head == first))
        isEmpty_abs();
    }
    // Note that if the prophecy is true,
    // then now we also have head = first
    {next, first, last, ret ⊩
     ∃x1, x2, y1, y2, z1, z2. Q2(x1, x2, y1, y2, z1, z2, _, _) ∧
     (first.ptr ≠ last.ptr ∧ next.ptr ≠ 0 ⇒ (x2 > first.count ∧
     ¬prophecy(l: head == first) ∧ Pre(ret[tid] ↦ ret) ∨
     prophecy(l: head == first) ∧ Post(ret[tid] ↦ 0) ∧
     x2 = first.count ∧ x1 = first.ptr ∧
     y1 = next.ptr ∧ y2 = next.count) ∧ emp)
     (first.ptr = last.ptr ∧ next.ptr ≠ 0 ⇒
     (x2 > first.count ∨ z2 > last.count ∨
     x2 = first.count ∧ x1 = first.ptr ∧
     y1 = next.ptr ∧ y2 = next.count ∧
     z1 = last.ptr ∧ z2 = last.count) ∧
     Pre(ret[tid] ↦ ret) ∧ emp) ∧
     (first.ptr = last.ptr ∧ next.ptr = 0 ⇒
     (x2 > first.count ∧ ¬prophecy(l: head == first) ∧ emp ∨
     x2 = first.count ∧ x1 = first.ptr ∧
     y1 = next.ptr ∧ y2 = next.count ∧
     z1 = last.ptr ∧ z2 = last.count ∧
     prophecy(l: head == first) ∧ Post(ret[tid] ↦ 1)))
    l: if (atomic_ld { head != first }) continue;
    {next, first, last, ret ⊩
     ∃x1, x2, y1, y2, z1, z2. Q2(x1, x2, y1, y2, z1, z2, _, _) ∧
     (first.ptr ≠ last.ptr ∧ next.ptr ≠ 0 ⇒
     Post(ret[tid] ↦ 0) ∧ emp) ∧
     (first.ptr = last.ptr ∧ next.ptr ≠ 0 ⇒ (z2 > last.count ∨
     x2 = first.count ∧ x1 = first.ptr ∧
     y1 = next.ptr ∧ y2 = next.count ∧
     z1 > last.ptr ∧ z2 > last.count) ∧ Pre(ret[tid] ↦ ret)) ∧
     (first.ptr = last.ptr ∧ next.ptr = 0 ⇒
     Post(ret[tid] ↦ 1))}
    if (first.ptr != last.ptr) {
      {next, first, last, ret ⊩ QueueInv ∧
       Post(ret[tid] ↦ 0) ∧ emp)
      return 0;
    }
    if (next.ptr == NULL) {
      {next, first, last, ret ⊩
       QueueInv ∧ Post(ret[tid] ↦ 1)}
      return 1;
    }
    {next, first, last, ret ⊩ Pre(ret[tid] ↦ ret) ∧
     ∃x1, x2, y1, y2, z1, z2. Q2(x1, x2, y1, y2, z1, z2, _, _) ∧
     first.ptr = last.ptr ∧ next.ptr ≠ 0 ∧ (z2 > last.count ∨
     x2 = first.count ∧ x1 = first.ptr ∧
     y1 = next.ptr ∧ y2 = next.count ∧
     z1 = last.ptr ∧ z2 = last.count) ∧ emp}
    CAS(tail, last, NodeRef(next.ptr, last.count+1));
  }
}
```