

Liveness-preserving atomicity abstraction

Alexey Gotsman¹ and Hongseok Yang²

¹ IMDEA Software Institute

² Queen Mary University of London

Abstract. Modern concurrent algorithms are usually encapsulated in libraries, and complex algorithms are often constructed using libraries of simpler ones. We present the first theorem that allows harnessing this structure to give compositional liveness proofs to concurrent algorithms and their clients. We show that, while proving a liveness property of a client using a concurrent library, we can soundly replace the library by another one related to the original library by a generalisation of a well-known notion of linearizability. We apply this result to show formally that lock-freedom, an often-used liveness property of non-blocking algorithms, is compositional for linearizable libraries, and provide an example illustrating our proof technique.

1 Introduction

Concurrent systems are usually expected to satisfy *liveness* properties [1], which, informally, guarantee that certain good events eventually happen. Reasoning about liveness in modern concurrent programs is difficult. Fortunately, the task can be simplified using reasoning methods that are able to exploit program structure. For example, concurrent algorithms are usually encapsulated in libraries and complex algorithms are often constructed using libraries of simpler ones. Thus, in reasoning about liveness of client code of a concurrent library, we would like to abstract from the details of a particular library implementation. This requires a notion of library abstraction that is able to specify the relevant liveness properties of the library.

Sound abstractions of concurrent libraries are commonly formalised by the notion of *linearizability* [11], which fixes a certain correspondence between the library and its abstract specification (the latter usually sequential, with methods implemented atomically). However, linearizability is not suitable for liveness. It takes into account finite computations only and does not restrict the termination behaviour of library methods when relating them to methods of an abstract specification. As a result, the linearizing specification loses most of the liveness properties of the library. For example, no linearizing specifications can specify that library methods always terminate or a method meant to acquire a resource may not always return a value signifying that the resource is busy. In this paper we propose a generalisation of linearizability that lifts the above limitations and allows specifying such properties (§3).

Building on our generalized notion of linearizability, we present a theorem that allows giving liveness proofs to concurrent algorithms that are compositional in their structure (Theorem 4, §4). Namely, we show that, while proving a liveness property of a client of a concurrent library, we can soundly replace the library by a simpler one related to the original library by our generalisation of linearizability. To our knowledge, this is the first result, either for safety or liveness, that allows exploiting linearizability in verifying concurrent programs. In particular, it enables liveness-preserving *atomicity abstraction*. When proving a liveness property of a client using a concurrent library, we can replace the library by its atomic abstract specification, and prove the liveness of the client with respect to this specification instead.

We further show that we can use existing tools for proving classical linearizability (e.g., [16, 15, 2]) to establish our generalisation. To this end, we identify a class of *linearization-closed* properties that, when satisfied by a library, are also satisfied by any other library linearizing it. This allows us to perform atomicity abstraction in two stages. We first use existing tools to establish the linearizability of a library to a coarse specification, sufficient only for proving safety properties of a client. We can then strengthen the specification for free with any linearization-closed liveness properties proved of the library implementation (Corollary 7, §5).

Finally, we demonstrate how our results can be used to give compositional proofs of *lock-freedom*, a liveness property often required of modern concurrent algorithms. In particular, we show (Theorem 9, §6) that lock-freedom is a compositional property for linearizable libraries: when proving it of an algorithm using a linearizable lock-free library, we can replace library methods by their atomic always-terminating specifications. Our formalisation also highlights a (perhaps surprising) fact that compositionality does not hold for a variant of lock-freedom that assumes fair scheduling. We demonstrate the resulting proof technique on a non-blocking stack by Hendler et al. [9] (§7).

2 Preliminaries

Programming language. We consider a simple language for heap-manipulating concurrent programs, where a program consists of a library implementing several methods and its client, given by a parallel composition of threads. Let Var be a set of variables, ranged over by x, y, \dots , and Method a set of method names, ranged over by m . For simplicity, all methods take one argument. The syntax of the language is given below:

$$\begin{aligned}
E &::= \mathbb{Z} \mid x \mid E + E \mid -E \mid \dots & C &::= c \mid m(E) \mid C; C \mid C + C \mid C^\circ \\
D &::= C; \text{return}(E) & & \text{(where } C \text{ does not include method calls } m(E)\text{)} \\
A &::= \{m = D, \dots, m = D\} \\
C(A) &::= \text{let } A \text{ in } C \parallel \dots \parallel C & & \text{(where all the methods called are defined in } A\text{)}
\end{aligned}$$

The language includes primitive commands c , method call $m(E)$, sequential composition $C; C'$, nondeterministic choice $C + C'$, and iteration C° . We use a non-standard notation C° here (instead of the usual Kleene star C^*), so as to emphasise that C° describes not just finite, but also infinite iterations of C . Both primitive commands and method calls $m(E)$ are assumed atomic. The primitive commands form the set PComm , and they include standard instructions, such as skip, variable assignment $x = E$, the update $[E] = E'$ of the heap cell E by E' , the read $x = [E]$ of the heap cell E into the variable x , and the assume statement $\text{assume}(E)$, filtering out states making $E = 0$. We point out that the standard constructs, such as loops and conditionals, can be defined in our language as syntactic sugar, with conditions translated using assume (see also §A).

Let us fix a program $C(A) = \text{let } A \text{ in } C_1 \parallel \dots \parallel C_n$ with the library definition $A = \{m = D_m \mid m \in M\}$. We let the signature of the library A be the set of the implemented methods: $\text{sig}(A) = M$. In the following we index threads in programs using the set of identifiers $\text{ThreadID} = \mathbb{N}$.

We restrict programs to ensure that the state of the client is disjoint from that of the library, and this state separation is respected by client operations and library routines. We note that this restriction is assumed by the standard notion of linearizability [11]; see §9 for discussion. Technically, we assume $\text{PComm} = \text{ClientPComm} \uplus \text{LibPComm}$ for some sets ClientPComm and LibPComm and require that all primitive commands in the client be from ClientPComm and those in the library from LibPComm . As formalised

below, the commands in ClientPComm and LibPComm can access only variables and heap locations belonging to the client and the library, respectively.

We also assume special variables $\text{retval}_t \in \text{Var}$ for each thread t in the program and $\text{param} \in \text{Var}$. The variable retval_t contains the result of the most recent method call and is implicitly updated by the return command. No commands in the program are allowed to modify retval_t explicitly, and the variable can only be read by C_t . The variable param is used to keep the values of parameters passed upon method calls and is implicitly updated by the call command. We assume that all the variables occurring in the expression E of a call command $m(E)$ are accessed only by the thread executing the command.

State model. Let CLoc and LLoc be disjoint sets representing heap locations that belong to the address spaces of the client and the library, respectively. We also assume $\text{Var} = \text{CVar} \uplus \text{LVar}$ for some sets CVar and LVar representing variables that belong to the client and the library, respectively. Let $\text{param} \in \text{LVar}$ and $\text{retval}_t \in \text{CVar}$, $t = 1..n$. We then define the set of program states State as follows:

$$\begin{aligned} \text{Loc} &= \text{CLoc} \uplus \text{LLoc} & \text{Val} &= \mathbb{Z} \uplus \text{CLoc} \uplus \text{LLoc} \\ \text{Stack} &= \text{Var} \rightarrow \text{Val} & \text{Heap} &= \text{Loc} \rightarrow \text{Val} & \text{State} &= \text{Stack} \times \text{Heap} \end{aligned}$$

A state in this model consists of a stack and a heap, both of which are total maps from variables or locations to values. Every location or variable is owned either by the library or by the client. We make this ownership explicit by defining two sets, CState for client states and LState for library states: $\text{CState} = (\text{CVar} \rightarrow \text{Val}) \times (\text{CLoc} \rightarrow \text{Val})$ and $\text{LState} = (\text{LVar} \rightarrow \text{Val}) \times (\text{LLoc} \rightarrow \text{Val})$. Also, we define three operations relating these sets to State: $\text{client} : \text{State} \rightarrow \text{CState}$, $\text{lib} : \text{State} \rightarrow \text{LState}$ and $\circ : \text{CState} \times \text{LState} \rightarrow \text{State}$. The first two are projections from states to client and library states, and are defined by restricting the domains of the stack and the heap: e.g., for $(s, h) \in \text{State}$ we have $\text{lib}(s, h) = (s|_{\text{LVar}}, h|_{\text{LLoc}})$. The \circ operator combines client and library states into program states: $(s_1, h_1) \circ (s_2, h_2) = (s_1 \uplus s_2, h_1 \uplus h_2)$. We lift client, lib and \circ to sets of states pointwise.

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 $(N, T, \text{start}, \text{end})$, consisting of the set of program positions N , the control-flow relation $T \subseteq N \times \text{Comm} \times N$, and the initial and final positions $\text{start}, \text{end} \in N$. The edges are annotated with commands from Comm, which are primitive commands, calls $m(E)$, or returns $\text{return}(E)$. Every command C in our language can be translated to a CFG in a standard manner (for completeness we provide the translation in §A). Hence, we can represent a program $C(A)$ by a collection of CFGs: the client command C_t for a thread t is represented by $(N_t, T_t, \text{start}_t, \text{end}_t)$, and the body D_m of a method m by $(N_m, T_m, \text{start}_m, \text{end}_m)$. We often view this collection of CFGs for $C(A)$ as a single graph consisting of two node sets $\text{CNode} = \uplus_{t=1}^n N_t$ and $\text{LNode} = \uplus_{m \in \text{sig}(A)} N_m$, and the edge set $T = \uplus_{t=1}^n T_t \uplus \uplus_{m \in \text{sig}(A)} T_m$. Finally, we define $\text{method} : \text{LNode} \rightarrow M$ as follows: $\text{method}(v) = m$ if and only if $v \in N_m$.

Program semantics. Programs in our semantics denote sets of *traces*, which are finite or infinite sequences of actions of the form

$$\varphi ::= (t, \text{Client}(c)) \mid (t, \text{Lib}(c)) \mid (t, \text{call } m(k)) \mid (t, \text{ret } m(k))$$

where $t \in \text{ThreadID}$, $c \in \text{PComm}$, $k \in \text{Val}$, and $m \in \text{Method}$. Each action corre-

Fig. 1 Operational semantics of the programming language. Here $\llbracket E \rrbracket s \in \text{Val}$ is the value of the expression E in the stack s . We denote with $g[x : y]$ the function that has the same value as the function g everywhere except x , where it has the value y . We remind the reader that T is the control-flow relation of $C(A)$.

$$\begin{array}{c}
\frac{(v, c, v') \in T \quad \theta' \in f_c(\theta)}{\text{pc}[t : v], \theta \xrightarrow{(t, \text{Client}(c))}_{C(A)} \text{pc}[t : v'], \theta'} \quad \frac{(v, m(E), v') \in T \quad \llbracket E \rrbracket s = k}{\text{pc}[t : v], \theta \xrightarrow{(t, \text{call } m(k))}_{C(A)} \text{pc}[t : \langle k, v', \text{start}_m \rangle], \theta} \\
\frac{(v, c, v') \in T \quad (s', h') \in f_c(s[\text{param} : k], h)}{\text{pc}[t : \langle k, v_0, v \rangle], (s, h) \xrightarrow{(t, \text{Lib}(c))}_{C(A)} \text{pc}[t : \langle s'(\text{param}), v_0, v' \rangle], (s', h')} \\
\frac{(v, \text{return}(E), v') \in T \quad \llbracket E \rrbracket (s[\text{param} : k]) = k'}{\text{pc}[t : \langle k, v_0, v \rangle], (s, h) \xrightarrow{(t, \text{ret } (\text{method}(v))(k'))}_{C(A)} \text{pc}[t : v_0], (s[\text{retval}_t : k'], h)}
\end{array}$$

sponds to a primitive command c executed by the client or the library (in which case we tag c with Client or Lib), a call to or a return from the library. We denote the sets of each kind of actions with ClientAct, LibAct, CallAct and RetAct, respectively, and we let CallRetAct be CallAct \cup RetAct and Act the set of all actions. Also, we write Trace for the set of all traces and adopt the standard notation: ε is the empty trace, $\tau(i)$ is the i -th action in the trace τ , and $|\tau|$ is the length of the trace τ ($|\tau| = \omega$ if τ is infinite).

Our semantics assumes an interpretation of every primitive command $c \in \text{PComm}$ as a transformer f_c of type $\text{LState} \rightarrow \mathcal{P}(\text{LState})$ if $c \in \text{LibPComm}$, or of type $\text{CState} \rightarrow \mathcal{P}(\text{CState})$ if $c \in \text{ClientPComm}$. In both cases, we lift this transformer to a function $f_c : \text{State} \rightarrow \mathcal{P}(\text{State})$ by transforming only the client or the library part of the input state:

$$\forall \theta \in \text{State}. f_c(\theta) \stackrel{\text{def}}{=} \begin{cases} \{\text{client}(\theta)\} \circ f_c(\text{lib}(\theta)), & \text{if } c \in \text{LibPComm}; \\ f_c(\text{client}(\theta)) \circ \{\text{lib}(\theta)\}, & \text{if } c \in \text{ClientPComm}. \end{cases} \quad (1)$$

For completeness, we provide the transformers for sample primitive commands in §A.

Let the set of **thread positions** be defined as follows: $\text{Pos} = \text{CNode} \cup (\text{Val} \times \text{CNode} \times \text{LNode})$. Elements in Pos describe the runtime status of a thread. A node $v \in \text{CNode}$ indicates that the thread is about to execute the client code coming right after the node v in its CFG. A triple $\langle k, v, v' \rangle$ means that the thread is at the program point v' in the code of a library method, k is the value of the param variable and v the return program point in the client code.

The semantics of the program $C(A)$ with threads $\{1, \dots, n\}$ is defined using a transition relation $\longrightarrow_{C(A)} : \text{Config} \times \text{Act} \times \text{Config}$, which transforms program configurations $\text{Config} = (\{1, \dots, n\} \rightarrow \text{Pos}) \times \text{State}$. The configurations are pairs of program counters and states, where a program counter defines the position of each thread in the program. The relation is defined by the rules in Figure 1. Note that, upon a method call, the actual parameter and the return point are saved as components in the new program position, and the method starts executing from the corresponding starting node of its CFG. The saved actual parameter is accessed whenever the library code reads the variable param, as modelled in the third rule for the library action. Upon a return, the return point is read from the current program counter, and the return value is written into the retval_t for the thread t executing the return command.

Our operational semantics induces the trace interpretation of programs $C(A)$. For a finite trace τ and $\sigma, \sigma' \in \text{Config}$ we write $\sigma \xrightarrow{\tau}^*_{C(A)} \sigma'$ if there exists a corresponding derivation of τ using \longrightarrow . Similarly, for an infinite trace τ and $\sigma \in \text{Config}$ we write $\sigma \xrightarrow{\tau}_{C(A)}^\omega$ to mean the existence of infinite τ -labelled computation from σ according

to our semantics. Let us denote with pc_0 the initial program counter $[1 : start_1, \dots, n : start_n]$. The trace semantics of $C(A)$ is defined as follows:

$$\llbracket C(A) \rrbracket = \{\tau \mid \exists \theta \in \text{State}. (pc_0, \theta) \xrightarrow{\tau} \omega_{C(A)} - \vee \exists \sigma \in \text{Config}. (pc_0, \theta) \xrightarrow{\tau} {}^*_{C(A)} \sigma\}.$$

Note that $\llbracket C(A) \rrbracket$ includes all finite or infinite traces (including non-maximal ones).

Client and library traces. In this paper we consider two special kinds of traces: *client traces*, which include only actions from $\text{CallRetAct} \cup \text{ClientAct}$, and *library traces*, which contain only actions from $\text{CallRetAct} \cup \text{LibAct}$. Given a trace $\tau \in \text{Trace}$, we can obtain the corresponding client $\text{client}(\tau)$ and library $\text{lib}(\tau)$ traces by projecting τ to the appropriate subsets of actions. We consider two further projections: $\text{visible}(\tau)$ projects τ to actions in ClientAct , and $\tau|_t$ to actions of thread t . We let CTrace be the set of all client traces and LTrace the set of all library traces.

Histories. We record interactions between the client and the library using *histories*, which are sequences of actions from $\text{CallRetAct} \cup \text{BlockAct}$, where $\text{BlockAct} = \{(t, \text{block}) \mid t \in \text{ThreadID}\}$. An action (t, block) means that thread t is suspended by the scheduler and is never scheduled again. We record block events because the liveness properties we are dealing with in this paper need to distinguish between method non-termination due to divergence and the one due to being suspended by the scheduler forever. Let History be the set of all histories.

Given a trace $\tau \in \text{Trace}$, we can construct a corresponding history $\text{history}(\tau)$ in two steps. First, for every thread t in τ , if the last action of the thread is an action in $\text{CallAct} \cup \text{LibAct}$, we insert (t, block) right after this action in τ , obtaining a sequence of actions τ' . The inserted (t, block) indicates that the thread t is suspended while running a library method and is never scheduled again. The history $\text{history}(\tau)$ is then obtained by projecting τ' to actions in $\text{CallRetAct} \cup \text{BlockAct}$.

3 Concurrent library semantics and linearizability

The correctness of concurrent libraries is usually defined using the notion of linearizability [11], which fixes a particular correspondence between the implementation and the specification of a library. We now define an analogue of this notion in our setting.

Definition 1. *The linearizability relation is a binary relation \sqsubseteq on histories defined as follows: $H \sqsubseteq H'$ if $\forall t \in \text{ThreadID}. H|_t = H'|_t$ (where $-|_t$ include block actions) and there is a bijection $\pi: \{1, \dots, |H|\} \rightarrow \{1, \dots, |H'|\}$ such that $\forall i. H(i) = H'(\pi(i))$ and*

$$\forall i, j. (i < j \wedge H(i) \in \text{RetAct} \wedge H(j) \in \text{CallAct}) \Rightarrow \pi(i) < \pi(j).$$

That is, the history H' linearizes the history H when it is a permutation of the latter preserving the order of non-overlapping method invocations and actions within threads. The duration of a method invocation is defined by the interval from the method call action to the corresponding return action (or to infinity if there is none). Our definition allows H and H' to be infinite, in contrast to the standard notion of linearizability which considers only finite histories.

To check if one library linearizes another, we need to define the set of histories a library can generate. We do this using the *most general client* of the library, which invokes any library methods in any order and with all possible parameters. As we show in Lemma 10(1), §B, the client we define here is indeed most general in the sense that its semantics includes all library behaviours generated by any other client in our programming language. Formally, consider a library $A = \{m = D_m \mid m \in M\}$.

The most general client $\text{MGC}_n(A)$ is the combination of CFGs for the library A and those for the client with n threads that repeatedly invoke library methods: the CFG for thread t is $(N_t, T_t, v_{\text{mgc}}^t, v_{\text{mgc}}^t)$ with $N_t = \{v_{\text{mgc}}^t\}$ and $T_t = \{(v_{\text{mgc}}^t, m(k), v_{\text{mgc}}^t) \mid m \in \text{sig}(A), k \in \text{Val}\}$. One can understand $\text{MGC}_n(A)$ as “let A in $C_{\text{mgc}}^1 \parallel \dots \parallel C_{\text{mgc}}^n$ ” where C_{mgc}^t repeatedly makes all possible method calls. Let $N_{\text{mgc}} = \bigsqcup_{t=1}^n N_t$.

To simplify our proofs, we define the semantics of $\text{MGC}_n(A)$ in a **library-local semantics**, where the program executes only on the library part of state. Namely, we consider a relation $\longrightarrow_{\text{MGC}_n(A)}: \text{LConfig} \times \text{Act} \times \text{LConfig}$ transforming configurations $\text{LConfig} = (\{1, \dots, n\} \rightarrow \text{LPos}) \times \text{LState}$, where $\text{LPos} = N_{\text{mgc}} \cup (\text{Val} \times N_{\text{mgc}} \times \text{LNode})$. The relation is defined as in Figure 1 with the rule for return commands replaced by the one that does not write the return value into retval_t (since this variable is not part of library states in LState):

$$\frac{(v, \text{return}(E), v') \in T \quad \llbracket E \rrbracket(s[\text{param} : k]) = k'}{\text{pc}[t : \langle k, v_0, v \rangle], (s, h) \xrightarrow{(t, \text{ret}(\text{method}(v))(k'))}_{\text{MGC}_n(A)} \text{pc}[t : v_0], (s, h)}$$

Let $\llbracket \text{MGC}_n(A) \rrbracket_{\text{lib}} \subseteq \text{LTrace}$ be the set of all traces generated by the program $\text{MGC}_n(A)$ in this semantics from any initial state in LState . We then define the set of all possible behaviours of the library A as the set of its traces $\llbracket A \rrbracket = \bigcup_{n \geq 1} \llbracket \text{MGC}_n(A) \rrbracket_{\text{lib}}$. This lets us lift the notion of linearizability to libraries as follows.

Definition 2. For libraries A_1 and A_2 with $\text{sig}(A_1) = \text{sig}(A_2)$ we say that A_2 **linearizes** A_1 , written $A_1 \sqsubseteq A_2$, if $\forall H_1 \in \text{history}(\llbracket A_1 \rrbracket). \exists H_2 \in \text{history}(\llbracket A_2 \rrbracket). H_1 \sqsubseteq H_2$.

Thus, A_2 linearizes A_1 if every behaviour of the latter under the most general client may be reproduced in a linearized form by the former.

It is instructive to compare our definition of linearizability with the classical one [11]. The original definition of linearizability between an implementation A_1 of a concurrent library and its specification A_2 considers only finite histories without block actions. It assumes that the specification A_2 is sequential, meaning that every method in it is implemented by an atomic always-terminating command. To deal with non-terminating method calls in A_1 when comparing the sets of histories generated by the two libraries, the original definition *completes* the histories of A_1 before the comparison: for every call action in the history without a corresponding return action, either the action is discarded or a corresponding return action with an arbitrary return value is added at some point in the history. Such an arbitrary completion of pending calls does not allow making any statements about the termination behaviour of the library in its specification. Furthermore, the fact that the definition considers any histories of A_2 consistent with the sequential specification makes it impossible to specify trace-based liveness properties satisfied by the library, e.g., that a method meant to acquire a resource may not always return a value signifying that the resource is busy.

Our definition lifts these limitations in a bid to enable compositional reasoning about liveness properties of concurrent libraries. Definitions 1 and 2 take into account infinite computations and do not require the specification A_2 to be sequential. Thus, they allow method calls in A_2 not to terminate and the execution of such methods to overlap with the executions of others. Our definitions require any method non-termination in the implementation to be reproducible in the specification. As we show in §5, by restricting the set of histories of the specification with fairness constraints, we can specify trace-based liveness properties. As we discuss in §8, our definition is more flexible than previous attempts at generalising linearizability to deal with method non-termination.

The classical notion of linearizability can also be expressed in our setting. We use this in §5 to harness existing linearizability checkers in reasoning about liveness properties. The linearizability of concurrent libraries according to the classical definition can be established using several logics and tools, e.g., based on separation logic [16, 15] or TVLA [2]. These logics and tools reduce proving linearizability to proving an invariant relating the states of the implementation and the sequential specification. They establish the validity of the invariant both on finite and infinite computations. Hence, it can be shown that they also establish linearizability in the sense of Definition 1. More precisely, assume a specification of the effect of every method m in a library A_1 given as a command $c_m; \text{return}(E_m)$ for some $c_m \in \text{PComm}$. Then the tools in [16, 15, 2] establish $A_1 \sqsubseteq A_2$, where $A_2 = \{m = (\text{skip}^\circ; c_m; \text{skip}^\circ; \text{return}(E_m)) \mid m \in \text{sig}(A_1)\}$. It is easy to show that this linearizability relation, when restricted to finite histories, is equivalent to the classical notion of linearizability [11].

Since the classical notion of linearizability does not specify the termination behaviour of the library, methods in the library A_2 above may diverge. The divergence can happen either before the method makes a change to the library state using c_m or after, as modelled by the two skip° statements. In fact, both of these cases are exhibited by practical concurrent algorithms. For example, the classical Treiber’s stack [14] and its variations [9] do not modify the state in the case of divergence, but Harris’s non-blocking linked list [8] does. History completion in the classical definition of linearizability models the effect of divergence at one of the skip° statements in our formalisation: discarding a pending call models the divergence at the first one, and completing a pending call with an arbitrary return models the divergence at the second. We are able to express the classical notion of linearizability in a uniform way because the specification A_2 above is not sequential: we allow non-terminating method invocations to overlap with others in the specification and, hence, do not need to complete them.

4 Atomicity abstraction

We now show how our notion of linearizability can be used to abstract an implementation of a library while reasoning about liveness properties of its client. Namely, we prove that replacing a library used by a client with its linearization leaves all the original client behaviours reproducible modulo the following notion of trace equivalence:

Definition 3. *Client traces $\tau, \tau' \in \text{CTrace}$ are **equivalent**, written $\tau \sim \tau'$, if there exists a bijection $\pi : \{1, \dots, |\tau|\} \rightarrow \{1, \dots, |\tau'|\}$ such that $(\forall i. \tau(i) = \tau'(\pi(i)))$ and*

$$\begin{aligned} & (\forall i, j. (\tau(i), \tau(j) \in \text{ClientAct} \wedge i < j) \Rightarrow \pi(i) < \pi(j)) \wedge \\ & (\forall i, j, t, a, a'. (\tau(i) = (t, a) \wedge \tau(j) = (t, a') \wedge i < j) \Rightarrow \pi(i) < \pi(j)). \end{aligned}$$

Two traces are equivalent if they are permutations of each other preserving the order of actions within threads and all client actions. Note that $\text{visible}(\tau) = \text{visible}(\tau')$ when $\tau \sim \tau'$. Hence, trace equivalence preserves any linear-time temporal property over trace projections to client actions. The following theorem states the desired abstraction result.

Theorem 4 (Abstraction). *Consider $C(A_1)$ and $C(A_2)$ such that $A_1 \sqsubseteq A_2$. Then*

$$\forall \tau_1 \in \llbracket C(A_1) \rrbracket. \exists \tau_2 \in \llbracket C(A_2) \rrbracket. \text{client}(\tau_1) \sim \text{client}(\tau_2) \wedge \text{history}(\tau_1) \sqsubseteq \text{history}(\tau_2).$$

Due to space constraints, we defer the proof of the theorem to §B.

Corollary 5. *If $A_1 \sqsubseteq A_2$, then $\text{visible}(\llbracket C(A_1) \rrbracket) \subseteq \text{visible}(\llbracket C(A_2) \rrbracket)$.*

According to Corollary 5, while reasoning about a client $C(A_1)$ of a library A_1 , we can soundly replace A_1 with a simpler library A_2 linearizing A_1 : if a linear-time liveness

property over client actions holds over $C(A_2)$, it will also hold over $C(A_1)$. In practice, we are usually interested in *atomicity abstraction*, a special case of the above transformation when methods in A_2 are implemented using atomic commands. In §6 we apply this technique to proving liveness properties of modern concurrent algorithms. Before this, however, we need to explain how to establish the required linearizability relation $A_1 \sqsubseteq A_2$. This is the subject of the next section.

5 Linearization-closed properties

As we explained in §3, existing tools can only prove linearizability relations $A_1 \sqsubseteq A_2$ such that A_2 has the form $\{m = (\text{skip}^\circ; c_m; \text{skip}^\circ; \text{return}(E_m)) \mid m \in \text{sig}(A_1)\}$ for some atomic commands c_m . The specification A_2 of the library A_1 is too coarse to prove a non-trivial liveness property of a client, as it allows methods to diverge and does not permit specifying liveness properties. If the library A_1 satisfies some liveness property (e.g., its method invocations not blocked by the scheduler always terminate), we would like to carry it over to A_2 to use in the proof of the client. This is, in fact, possible for a certain class of properties.

Definition 6. A property $P \subseteq \text{History}$ over histories is *linearization-closed* if $\forall H, H'. (H \in P \wedge H \sqsubseteq H') \Rightarrow H' \in P$.

Intuitively, linearization-closed properties should be formulated in terms of pairs of call and return actions and not in terms of separate actions, as the latter can be rearranged during linearization. For example, the property “method invocations that are not blocked by the scheduler always terminate” is linearization-closed. In contrast, the property “a $(t, \text{ret } m(0))$ action is always followed by a $(t', \text{ret } m(1))$ action” is not.

Corollary 7. Let $P \subseteq \text{History}$ be linearization-closed, $A_1 \sqsubseteq A_2$, and $\text{history}(\llbracket A_1 \rrbracket) \subseteq P$. Then $\text{visible}(\llbracket C(A_1) \rrbracket) \subseteq \text{visible}(\llbracket C(A_2) \rrbracket) \cap \{\tau \mid \text{history}(\tau) \in P\}$.

This corollary of Theorem 4, improving on Corollary 5, allows us to perform atomicity abstraction in two stages. We start with the coarse refinement $A_1 \sqsubseteq A_2$ established by tools for classical linearizability. If a liveness property P over histories holds of the implementation A_1 and is linearization-closed, the trace set of the specification A_2 can be shrunk by removing those violating P . To convert $C(A_2)$ into a program with the trace set $\llbracket C(A_2) \rrbracket \cap \{\tau \mid \text{history}(\tau) \in P\}$ we can use standard automata-theoretic techniques from model checking: we represent P by an automaton and construct a synchronous product of $C(A_2)$ and the automaton. See [17, 5, 7] for more details.

6 Compositional liveness proofs for concurrent algorithms

Lock-freedom. We now illustrate how Corollary 7 can be used to perform compositional proofs of liveness properties of *non-blocking* concurrent algorithms [10]. These complicated algorithms employ synchronisation techniques alternative to the usual lock-based mutual exclusion and typically provide high-performance concurrent implementations of data structures, such as stacks, queues, linked lists, and hash tables (see, for example, the `java.util.concurrent` library).

Out of all properties used to formulate progress guarantees for such algorithms, we concentrate on *lock-freedom*, as the one most often used and most difficult to prove. Informally, an algorithm implementing operations on a concurrent data structure is considered lock-free if from any point in a program’s execution, some thread is guaranteed to complete its operation. Thus, lock-freedom ensures the absence of livelock, but not

Fig. 2 A non-blocking stack implementation

```
struct Node {
    value_t data;
    Node *next;
};

Node *S;
int collision[SIZE];

void push(value_t v) {
    Node *t, *x;
    x = new Node();
    x->data = v;
    while (1) {
        t = S; x->next = t;
        if (CAS(&S,t,x))
            return;
        elim();
    }
}

void init() { S = NULL; }
void elim() { // Elimination scheme
    // ...
    int pos = GetPos(); // 0 ≤ pos ≤ SIZE-1
    int hisId = collision[pos];
    while (!CAS(&collision[pos],hisId,MYID))
        hisId = collision[pos];
    // ...
}

value_t pop() {
    Node *t, *x;
    while (1) {
        t = S;
        if (t == NULL) return EMPTY;
        x = t->next;
        if (CAS(&S,t,x)) return t->data;
        elim();
    }
}
```

starvation. The formal definition is as follows.

Definition 8. A library A is **lock-free** if for any $t \in \text{ThreadID}$, the set of its histories $\text{history}([A])$ satisfies $\text{LF} = \square \diamond (_, \text{ret } _) \vee \square (\text{call } _) \Rightarrow \diamond ((t, \text{block}) \vee (t, \text{ret } _))$.

Here we use linear temporal logic (LTL) over histories, with predicates over actions as atomic propositions; \square and \diamond are the standard operators “always” and “eventually” and $_$ stands for an irrelevant existentially quantified value. The property formalises the informal condition that some operation always complete by requiring that either some operation returns infinitely often (for the case when the client calls infinitely many operations), or every operation that has not been suspended by the scheduler forever returns (for the case when the client calls only finitely many operations).

Note that the semantics of §2 allows for unfair schedulers that suspend some threads and never resume them again. A crucial requirement in the definition of lock-freedom is that the property has to be satisfied under such schedulers: the threads that do get scheduled have to make progress even if others are suspended. In fact, formalising lock-freedom is the main reason for recording block actions in histories in this paper.

Lock-freedom of some algorithms, including Treiber’s stack, can be proved automatically [7].

Example. Consider the algorithm in Figure 2, ignoring the `elim` function and calls to it for now. For readability, the example is presented in C, rather than in our minimalistic language. It is a simple non-blocking implementation of a concurrent stack due to Treiber [14]. A client using the implementation can call several push or pop operations concurrently. To ensure the correctness of the algorithm, we assume that pop does not reclaim the memory taken by the deleted node [10]. The stack is stored as a linked list, and is updated by compare-and-swap (CAS) instructions. 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. The operations on the stack are implemented as follows. The function

`init` initialises the data structure. The push operation (i) allocates a new node `x`; (ii) reads the current value of the top-of-the-stack pointer `S`; (iii) makes the next field of the newly created node point to the read value of `S`; and (iv) atomically updates the top-of-the-stack pointer with the new value `x`. If the pointer has changed between (ii) and (iv) and has not been restored to its initial value, the CAS fails and the operation is restarted. The pop operation is implemented in a similar way.

Note that a push or pop operation of Treiber’s stack may diverge if other threads are continually modifying `S`: in this case the CAS instruction may always fail, which will cause the operation to restart continually. However, the algorithm is lock-free: if push and pop execute concurrently, some operation will always terminate.

Compositionality of lock-freedom. It is easy to check that the property LF in Definition 8 is linearization-closed. Thus, if A is a lock-free library and is linearized by a specification $\{m = (\text{skip}^\circ; c_m; \text{skip}^\circ; \text{return}(E_m)) \mid m \in \text{sig}(A)\}$, proving a liveness property of a client of A can be simplified if we replace A by the specification and consider only traces τ of the client such that $\text{history}(\tau)$ satisfies LF. As we now show, in the case when the client is itself a concurrent algorithm being proved lock-free, we can strengthen this result: we can assume a specification of A where every method terminates in all cases. We thus prove that lock-freedom is a compositional property of linearizable libraries.

To simplify presentation, we have not considered nested libraries, which makes the direct formulation of this result impossible. Instead, we rely on the reduction in [7], which says that an algorithm is lock-free if and only if any number of its operations running in parallel terminate, i.e., do not have infinite traces. (We repeat the reduction in Theorem 15, §C.) Thus, it suffices to prove the result for the termination of the client.

Theorem 9. *Let M be a set of method names. Consider libraries*

$$A_1 = \{m = D_m \mid m \in M\}, \quad A_2 = \{m = (c_m; \text{return}(E_m)) \mid m \in M\}, \\ A_3 = \{m = (\text{skip}^\circ; c_m; \text{skip}^\circ; \text{return}(E_m)) \mid m \in M\},$$

where c_m ’s are atomic commands, A_1 is lock-free, and $A_1 \sqsubseteq A_3$. If $\llbracket C(A_2) \rrbracket$ does not have infinite traces, then neither does $\llbracket C(A_1) \rrbracket$.

The proof is given in §C. Given the above reduction, the theorem implies that a concurrent algorithm using A_1 is lock-free if it is lock-free when it uses A_2 instead.

We note that some concurrent algorithms rely on locks, while satisfying lock-freedom under the assumption that the scheduler is fair [10]. Perhaps surprisingly, Theorem 9 does not hold in this case: lock-freedom under fair scheduling is not a compositional property. The intuitive reason is as follows: we can replace A_3 with A_2 in Theorem 9 because the effect of operations of A_3 diverging at one of the skip° statements is already covered by the possible unfairness of the scheduler. This reasoning becomes invalid once the scheduler is fair. We provide a counterexample in §C.

7 Example

Theorem 9 allows giving proofs of lock-freedom to non-blocking concurrent algorithms that are compositional in the structure of the algorithms, as we now illustrate. As an example, we consider an improvement of Treiber’s stack proposed by Hendler, Shavit, and Yerushalmi (HSY), which performs better in the case of higher contention among threads [9]. Figure 2 shows an adapted and abridged version of the algorithm. The implementation combines Treiber’s stack with a so-called elimination scheme, implemented by the function `elim` (partially elided). A push or a pop operation first tries

to modify the stack as in Treiber’s algorithm, by doing a CAS to change the shared top-of-the-stack pointer. If the CAS succeeds, the operation terminates. If the CAS fails (because of interference from another thread), the operation backs off to the elimination scheme. If this scheme fails, the whole operation is restarted.

The elimination scheme works on data structures that are separate from the list implementing the stack and, hence, can be considered as a library used by the HSY stack with the only method `elim`. The idea behind the scheme is that two contending push and pop operations can eliminate each other without modifying the stack if pop returns the value that push is trying to insert. An operation determines the existence of another operation it could eliminate itself with by selecting a random slot `pos` in the `collision` array, and atomically reading that slot and overwriting it with its thread identifier `MYID`. The algorithm implements the atomic read-and-write operation on the `collision` array in a lock-free fashion using CAS. The identifier of another thread read from the array can be subsequently used to perform elimination. The corresponding code does not affect the lock-freedom of the algorithm and is elided in Figure 2.

According to Theorem 9, to prove the lock-freedom of the HSY stack, it is sufficient to prove (i) the lock-freedom of the push and pop with a call to `elim` replaced by its atomic always-terminating specification; and (ii) the lock-freedom and linearizability of the `elim` method. The former is virtually identical to the proof of lock-freedom of Treiber’s stack, since `elim` acts on data structures disjoint from those of the stack. Informally, this proof is done as follows (see [7] for a detailed formal proof). It is sufficient to check the termination of the program consisting of a parallel composition of an arbitrary number of threads each executing one push or pop operation. For such a program, we have two facts. First, no thread executes a successful CAS in push or pop infinitely often. This is because once the CAS succeeds, the corresponding while-loop terminates. Second, the while-loop in an operation terminates if no other thread executes a successful CAS in push or pop infinitely often. This is because the operation does not terminate only when its CAS always fails, which requires the other threads to execute the CASes infinitely often. From these two facts, the termination of each thread follows.

The lock-freedom of the `elim` method can be proved in the same way and its linearizability can be proved using existing methods [15, 16]. This completes the proof of lock-freedom of the HSY stack. We have thus decomposed the proof of a complicated non-blocking algorithm with two nested loops into proofs of two simple algorithms.

8 Related work

Filipović et al. [6] have previously characterised linearizability in terms of observational refinement (technically, their result is similar to our Lemma 11, §B). They did not consider infinite computations and treated non-terminating methods approximately; thus, they could not handle liveness properties. Also, the work of Filipović et al. did not justify any compositional proof methods, as we have done in Theorem 4.

Petrank et al. [13] were the first to observe that lock-freedom is compositional, as we prove in Theorem 9. However, they argued this property only informally (and for so-called bounded version of lock-freedom, which is a safety and not a liveness property). As a result, they have missed an important requirement that library methods be linearizable.

Burckhardt et al. [4] have attempted to generalise linearizability on finite histories to the case of non-terminating method calls. However, their definition is too restrictive,

as it requires any non-termination in the library implementation to be reproducible in the *sequential* specification of the library. This requirement is not satisfied on infinite traces by common lock-free algorithms, where some methods may diverge while others make progress. Additionally, their definition considers any library where methods may modify the library state before diverging (e.g., [8]) as non-linearizable. We provide a more flexible definition.

Atomicity refinement is a well-known method for formal development of concurrent programs [3, 12], which allows refining an atomic specification to a concurrent implementation preserving the properties of specification. As atomicity refinement and abstractions are duals of each other, our results can also be used in the context of formal program development.

9 Conclusion

The main conceptual point of this paper is that proofs about modern concurrent programs can be made easier if they are constructed compositionally according to the program structure. Technically, this compositionality can be achieved using appropriate notions of abstractions for concurrent components and corresponding abstraction theorems, such as our Theorem 4. We have demonstrated this for a class of non-trivial properties and programs. In the future, we intend to investigate similar compositionality results in settings more flexible than the one considered here. For example, unlike in our setting, the library and the client usually reside in the same address space and may transfer the ownership of memory areas at calls and returns; they can interact via call-backs or calls to a third library, such as the memory allocator; finally, subtle interactions can arise from the concurrent program running on a weak memory model.

References

1. B. Alpern and F. B. Schneider. Defining liveness. *Inf. Process. Lett.*, 21(4):181–185, 1985.
2. D. Amit, N. Rinetzky, T. W. Reps, M. Sagiv, and E. Yahav. Comparison under abstraction for verifying linearizability. In *CAV*, 2007.
3. R.-J. Back. On correct refinement of programs. *JCSS*, 1981.
4. S. Burckhardt, C. Dern, M. Musuvathi, and R. Tan. Line-up: A complete and automatic linearizability checker. In *PLDI*, 2010.
5. B. Cook, A. Gotsman, A. Podelski, A. Rybalchenko, and M. Y. Vardi. Proving that programs eventually do something good. In *POPL*, 2007.
6. I. Filipović, P. O’Hearn, N. Rinetzky, and H. Yang. Abstraction for concurrent objects. In *ESOP*, 2009.
7. A. Gotsman, B. Cook, M. Parkinson, and V. Vafeiadis. Proving that non-blocking algorithms don’t block. In *POPL*, 2009.
8. T. Harris. A pragmatic implementation of non-blocking linked lists. In *DISC*, 2001.
9. D. Hendler, N. Shavit, and L. Yerushalmi. A scalable lock-free stack algorithm. In *SPAA*, 2004.
10. M. Herlihy and N. Shavit. *The art of multiprocessor programming*. 2008.
11. M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *TOPLAS*, 12:463–492, 1990.
12. C. Jones. Splitting atoms safely. *TCS*, 2007.
13. E. Petrank, M. Musuvathi, and B. Steensgaard. Progress guarantee via bounded lock-freedom. In *PLDI*, 2009.
14. R. K. Treiber. Systems programming: Coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center, 1986.
15. V. Vafeiadis. Modular fine-grained concurrency verification. PhD Thesis. Technical Report UCAM-CL-TR-726, University of Cambridge, 2008.
16. V. Vafeiadis. Automatically proving linearizability. In *CAV*, 2010.
17. M. Y. Vardi. Verification of concurrent programs—the automata-theoretic framework. *Annals of Pure and Applied Logic*, pages 79–98, 1991.

A Additional definitions for the programming language

Semantics of typical primitive commands. We consider the following sample primitive commands:

$$\text{skip}_i, (x = E)_i, (x = [E])_i, ([E] = E')_i, \text{assume}(E)_i,$$

where the subscript $i \in \{\text{client}, \text{lib}\}$ indicates whether the command is for the client code or for the library code. We assume that if the subscript i of a command is client, the command uses only client variables in CVar and make a similar assumption for the case $i = \text{lib}$.

For these commands, we define corresponding transition relations \rightsquigarrow in Figure 3. If a command has the client subscript, this relation transforms client states in CState, and if the commands has the lib subscript, the relation changes library states in LState. Using these transition relations, we then define f_c for the above commands c as follows:

$$f_c(\theta) = \bigcup \{\theta' \mid c, \theta \rightsquigarrow \theta'\}.$$

Here θ is a client or library state depending on whether c is a client command or a library command.

Definition of loops and conditionals. Let $!E$ be the C-language style negation of an expression E , so that $\llbracket !E \rrbracket s = 1$ if $\llbracket E \rrbracket s = 0$, and $\llbracket E \rrbracket s = 0$ for all the other cases. The standard commands for conditionals and loops are defined in our language as follows:

$$\begin{aligned} (\text{if}_i E \text{ then } C_1 \text{ else } C_2) &= (\text{assume}(E)_i; C_1) + (\text{assume}(!E)_i; C_2), \\ (\text{while}_i E \text{ do } C) &= (\text{assume}(E)_i; C)^\circ; \text{assume}(!E)_i, \end{aligned}$$

where i is client or lib, depending on which part of the code the command belongs to.

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 $(\{\text{start}, \text{end}\}, \{(\text{start}, c, \text{end})\}, \text{start}, \text{end})$.
2. Assume C_1 and C_2 have CFGs $(N_1, T_1, \text{start}_1, \text{end}_1)$ and $(N_2, T_2, \text{start}_2, \text{end}_2)$, respectively. Then $C_1; C_2$ has the CFG

$$(N_1 \cup N_2, T_1 \cup T_2 \cup \{(\text{end}_1, \text{skip}_i, \text{start}_2)\}, \text{start}_1, \text{end}_2).$$

3. Assume C_1 and C_2 have CFGs $(N_1, T_1, \text{start}_1, \text{end}_1)$ and $(N_2, T_2, \text{start}_2, \text{end}_2)$, respectively. Then $C_1 + C_2$ has the CFG

$$\begin{aligned} (N_1 \cup N_2 \cup \{\text{start}, \text{end}\}, T_1 \cup T_2 \cup \{(\text{start}, \text{skip}_i, \text{start}_1), (\text{start}, \text{skip}_i, \text{start}_2), \\ (\text{end}_1, \text{skip}_i, \text{end}), (\text{end}_2, \text{skip}_i, \text{end})\}, \text{start}, \text{end}). \end{aligned}$$

Fig. 3 Transition relation for sample primitive commands. Here (s_C, h_C) is a client state in CState and (s_L, h_L) a library state in LState.

$$\begin{aligned} &\text{skip}_{\text{client}}, (s_C, h_C) \rightsquigarrow (s_C, h_C) \\ &(x = E)_{\text{client}}, (s_C, h_C) \rightsquigarrow (s_C[x : \llbracket E \rrbracket s_C], h_C) \\ &(x = [E])_{\text{client}}, (s_C, h_C) \rightsquigarrow (s_C[x : h_C(\llbracket E \rrbracket s_C)], h_C) && (\text{when } \llbracket E \rrbracket s_C \in \text{CLoc}) \\ &([E] = E')_{\text{client}}, (s_C, h_C) \rightsquigarrow (s_C, h_C[\llbracket E \rrbracket s_C : \llbracket E' \rrbracket s_C]) && (\text{when } \llbracket E \rrbracket s_C \in \text{CLoc}) \\ &(\text{assume}(E))_{\text{client}}, (s_C, h_C) \rightsquigarrow (s_C, h_C) && (\text{when } \llbracket E \rrbracket s_C \neq 0) \\ \\ &\text{skip}_{\text{lib}}, (s_L, h_L) \rightsquigarrow (s_L, h_L) \\ &(x = E)_{\text{lib}}, (s_L, h_L) \rightsquigarrow (s_L[x : \llbracket E \rrbracket s_L], h_L) \\ &(x = [E])_{\text{lib}}, (s_L, h_L) \rightsquigarrow (s_L[x : h_L(\llbracket E \rrbracket s_L)], h_L) && (\text{when } \llbracket E \rrbracket s_L \in \text{LLoc}) \\ &([E] = E')_{\text{lib}}, (s_L, h_L) \rightsquigarrow (s_L, h_L[\llbracket E \rrbracket s_L : \llbracket E' \rrbracket s_L]) && (\text{when } \llbracket E \rrbracket s_L \in \text{LLoc}) \\ &(\text{assume}(E))_{\text{lib}}, (s_L, h_L) \rightsquigarrow (s_L, h_L) && (\text{when } \llbracket E \rrbracket s_L \neq 0) \end{aligned}$$

Fig. 4 Client-local semantics

$$\begin{array}{c}
 \frac{(v, c, v') \in T \quad \theta' \in f_c(\theta)}{\text{pc}[t : v], \theta \xrightarrow{(t, \text{Client}(c))}_{C(\cdot)} \text{pc}[t : v'], \theta'} \\
 \\
 \frac{(v, m(E), v') \in T \quad \llbracket E \rrbracket s = k}{\text{pc}[t : v], (s, h) \xrightarrow{(t, \text{call } m(k))}_{C(\cdot)} \text{pc}[t : \langle v', m \rangle], (s, h)} \\
 \\
 \text{pc}[t : \langle v, m \rangle], (s, h) \xrightarrow{(t, \text{ret } m(k))}_{C(\cdot)} \text{pc}[t : v], (s[\text{retval}_t : k], h)
 \end{array}$$

4. Assume C has a CFG $(N, T, \text{start}, \text{end})$. Then C° has the CFG

$$(N, T \cup \{(\text{end}, \text{skip}_i, \text{start})\}, \text{start}, \text{end}).$$

As before, i is client or lib, depending on which part of the code the command belongs to.

B Proof of Theorem 4

We first present an outline of the proof of Theorem 4; proofs of the lemmas below are given in the following sections. To prove the theorem, we need to be able to transform traces of $C(A_1)$ into traces of $C(A_2)$, which uses a different implementation of the library. We perform this transformation with the aid of a *client-local semantics*, which produces all behaviours of C regardless of the particular library implementation it uses. Let n be the number of threads in C . The semantics is defined by a transition relation $\longrightarrow_{C(\cdot)}: \text{CConfig} \times \text{Act} \times \text{CConfig}$ in Figure 4, which transforms configurations $\text{CConfig} = (\{1, \dots, n\} \rightarrow \text{CPos}) \times \text{CState}$ for $\text{CPos} = \text{CNode} \cup (\text{CNode} \times \text{Method})$. Note that, upon a call, we do not execute library actions, but just save the return point v' and the method m called in a thread position of the form $\langle v', m \rangle$. The semantics allows for an arbitrary library behaviour: we do not assign the value of the parameter to param at a call and let library methods return with an arbitrary value. We define the client-local semantics of C as the set $\llbracket C(\cdot) \rrbracket \subseteq \text{CTrace}$ of all traces generated using $\longrightarrow_{C(\cdot)}$ from any initial configuration in CConfig .

We thus have three semantics related to a program $C(A)$ used in the proof: the standard semantics $\llbracket C(A) \rrbracket \subseteq \text{Trace}$, the client-local semantics $\llbracket C(\cdot) \rrbracket \subseteq \text{CTrace}$, and the library-local semantics $\llbracket A \rrbracket \subseteq \text{LTrace}$. They are defined by the corresponding transition relations $\longrightarrow_{C(A)}$, $\longrightarrow_{C(\cdot)}$ and $\longrightarrow_{\text{MGC}_n(A)}$.

For a trace τ we define its *core history* $\text{corehistory}(\tau)$ as the projection of τ to actions in CallRetAct . We denote with CoreHistory the set of all core histories. A client trace $\tau \in \text{CTrace}$ is *well-formed* if for all $t \in \text{ThreadID}$, every call action $(t, \text{call } m(k))$ in the trace $\tau|_t$ is either the last one in the trace or is immediately followed by the corresponding return action $(t, \text{ret}(k') m)$, i.e., if every such projection behaves sequentially. The definition of well-formed core histories is the same.

We first state a lemma showing that a trace of $C(A)$ generates two traces in the client-local and library-local semantics, and any two such traces agreeing on the core history can be combined into a valid trace of $C(A)$.

Lemma 10 (Decomposition). 1. $\forall \tau \in \llbracket C(A) \rrbracket. \text{client}(\tau) \in \llbracket C(\cdot) \rrbracket \wedge \text{lib}(\tau) \in \llbracket A \rrbracket.$
 2. $\forall \eta \in \llbracket C(\cdot) \rrbracket. \forall \xi \in \llbracket A \rrbracket. \text{corehistory}(\eta) = \text{corehistory}(\xi) \Rightarrow$
 $\exists \tau \in \llbracket C(A) \rrbracket. \text{client}(\tau) = \eta \wedge \text{lib}(\tau) = \xi.$

The following lemma presents the core of the trace transformation used to generate τ_2 out of τ_1 in Theorem 4: it shows that a client trace can be transformed into an equivalent one with a given history linearizing the history of the original one.

Lemma 11 (Rearrangement). *Consider well-formed core histories $H, H' \in \text{CoreHistory}$ such that $H \sqsubseteq H'$. For all well-formed client traces $\tau \in \text{CTrace}$ such that $\text{corehistory}(\tau) = H$ there exists $\tau' \in \text{CTrace}$ such that $\tau \sim \tau' \wedge \text{corehistory}(\tau) = H'$.*

Finally, the following lemma shows that the trace resulting from the transformation in Lemma 11 is derivable in the client-local semantics if the original one is.

Lemma 12. *The set $\llbracket C(\cdot) \rrbracket$ is closed under trace equivalence.*

Proof of Theorem 4. Take $\tau_1 \in \llbracket C(A_1) \rrbracket$. By Lemma 10(1),

$$\text{client}(\tau_1) \in \llbracket C(\cdot) \rrbracket \wedge \text{lib}(\tau_1) \in \llbracket A \rrbracket.$$

Let $H_1 = \text{history}(\tau_1)$. Then, $H_1 \in \text{history}(\llbracket A_1 \rrbracket)$. Furthermore, since $A_1 \sqsubseteq A_2$, there exists a history $H_2 \in \text{history}(\llbracket A_2 \rrbracket)$ such that $H_1 \sqsubseteq H_2$. Then $\text{corehistory}(H_1) \sqsubseteq \text{corehistory}(H_2)$. Applying Lemma 11, we can construct a trace $\eta \in \text{CTrace}$ such that

$$\text{client}(\tau_1) \sim \eta \wedge \text{corehistory}(\eta) = \text{corehistory}(H_2).$$

By Lemma 12, $\eta \in \llbracket C(\cdot) \rrbracket$. Since $H_2 \in \text{history}(\llbracket A_2 \rrbracket)$, there exists a library trace $\xi \in \llbracket A_2 \rrbracket$ such that $\text{history}(\xi) = H_2$. Then $\text{corehistory}(\eta) = \text{corehistory}(\xi)$. By Lemma 10(2), for some $\tau_2 \in \llbracket C(A_2) \rrbracket$ we have $\eta = \text{client}(\tau_2)$ and $\xi = \text{lib}(\tau_2)$. Thus,

$$\text{client}(\tau_1) \sim \text{client}(\tau_2) \wedge \text{history}(\tau_2) = \text{history}(\text{lib}(\tau_2)) = H_2,$$

which implies that τ_2 is what we are looking for. \square

B.1 Proof of Lemma 10

Before proving the lemma, we introduce some auxiliary notation. For a state $\theta = (s, h)$, $x \in \text{Var}$ and $u \in \text{Val}$ we let $\theta[x : u] = (s[x : u], h)$ and $\theta(x) = s(x)$.

We define functions $\text{client} : \text{Pos} \rightarrow \text{CPos}$ and $\text{lib}_t : \text{Pos} \rightarrow \text{LPos}$, $t = 1..n$ that compute thread positions in the client-local and library-local semantics corresponding to a position in the complete program: $\text{client}(v) = v$, for $v \in \text{CNode}$; $\text{client}(\langle k, v', v \rangle) = \langle v', \text{method}(v) \rangle$ for $v' \in \text{CNode}$ and $v \in \text{LNode}$; $\text{lib}_t(v) = v_{\text{mgc}}^t$ for $v \in \text{CNode}$; $\text{lib}_t(\langle k, v', v \rangle) = \langle k, v_{\text{mgc}}^t, v \rangle$ for $v' \in \text{CNode}$ and $v \in \text{LNode}$. We then lift client and lib to program counters as follows: $(\text{client}(\text{pc}))(t) = \text{client}(\text{pc}(t))$ and $(\text{lib}(\text{pc}))(t) = \text{lib}_t(\text{pc}(t))$. Finally, we lift them to configurations: $\text{client}(\text{pc}, \theta) = (\text{client}(\text{pc}), \text{client}(\theta))$ and $\text{lib}(\text{pc}, \theta) = (\text{lib}(\text{pc}), \text{lib}(\theta))$.

We define a partial operation $\circ : \text{CPos} \times \text{LPos} \rightarrow \text{Pos}$ that combines thread positions in the client-local and library-local semantics to obtain a position in the complete program: $v \circ v_{\text{mgc}}^t = v$ for $v \in \text{CNode}$; $\langle v, m \rangle \circ \langle k, v_{\text{mgc}}^t, v' \rangle = \langle k, v, v' \rangle$ for $v \in \text{CNode}$ and $v' \in \text{LNode}$ such that $\text{method}(v') = m$; all other combinations are undefined. We then lift \circ to program counters pointwise and to configurations as follows: $(\text{pc}_1, \theta_1) \circ (\text{pc}_2, \theta_2) = (\text{pc}_1 \circ \text{pc}_2, \theta_1 \circ \theta_2)$. Note that for all $\sigma \in \text{State}$ we have $\text{client}(\sigma) \circ \text{lib}(\sigma) = \sigma$.

We now prove each of the two cases in the lemma separately.

Case 1. Consider $\tau \in \llbracket C(A) \rrbracket$ and let $\eta = \text{client}(\tau)$ and $\xi = \text{lib}(\tau)$. We need to show that $\eta \in \llbracket C(\cdot) \rrbracket$ and $\xi \in \llbracket A \rrbracket$. Since $\tau \in \llbracket C(A) \rrbracket$, there exists a τ -labelled computation sequence that starts from some initial configuration $\sigma_0 \in \text{Config}$ and follows the relation $\longrightarrow_{C(A)}$. Let n be the number of threads in $C(A)$. From the computation sequence of τ , we construct the computation sequence of η using $\longrightarrow_{C(\cdot)}$ and that

of ξ with $\longrightarrow_{\text{MGC}_n(A)}$. Our construction first considers every finite prefix τ_1 of τ and builds client-local and library-local computation sequences for τ_1 . Then we construct the desired computation sequences for η and ξ as the limits of these sequences (our construction is such that this limit exists). The following claim lies at the core of our construction:

Consider a finite prefix τ_1 of τ and a configuration $\sigma \in \text{Config}$ such that

$$\begin{aligned} \sigma_0 \xrightarrow{C(A)}^* \sigma \wedge \text{client}(\sigma_0) \xrightarrow{C(\cdot)}^* \text{client}(\sigma) \\ \wedge \text{lib}(\sigma_0) \xrightarrow{\text{MGC}_n(A)}^* \text{lib}(\sigma). \end{aligned}$$

If $\tau = \tau_1 \varphi \tau'_2$ for some action φ and trace τ'_2 , and $\sigma \xrightarrow{C(A)} \sigma'$ for some σ' , we have that

$$\text{client}(\sigma_0) \xrightarrow{C(\cdot)}^* \text{client}(\sigma') \wedge \text{lib}(\sigma_0) \xrightarrow{\text{MGC}_n(A)}^* \text{lib}(\sigma').$$

To prove the claim, we assume $\tau_1, \varphi, \tau'_2, \sigma, \sigma'$ satisfying the assumptions in the claim. There are four cases:

- $\varphi \in \text{ClientAct}$. In this case, there exist a thread $t \in \{1, \dots, n\}$, client nodes $v, v' \in \text{CNode}$, a client command $c \in \text{ClientPComm}$, and states $\theta, \theta' \in \text{State}$, such that (v, c, v') is in the control-flow relation of $C(A)$ and

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

Since c is a client operation, it follows from (1) in § 2 that

$$\text{client}(\theta') \in \text{client}(f_c(\theta)) = f_c(\text{client}(\theta)).$$

This in turn implies

$$\text{client}(\text{pc}[t : v], \text{client}(\theta)) \xrightarrow{C(\cdot)} \text{client}(\text{pc}[t : v'], \text{client}(\theta')).$$

Since $v, v' \in \text{CNode}$, we have

$$\text{client}(\text{pc}[t : v]) = \text{client}(\text{pc}[t : v]) \wedge \text{client}(\text{pc}[t : v']) = \text{client}(\text{pc}[t : v']).$$

Hence, the transition above is equivalent to $\text{client}(\sigma) \xrightarrow{C(\cdot)} \text{client}(\sigma')$, which implies that

$$\text{client}(\sigma_0) \xrightarrow{C(\cdot)}^* \text{client}(\sigma').$$

For the library computation sequence, we use (1) again and infer that

$$\text{lib}(\theta') \in \text{lib}(f_c(\theta)) = \{\text{lib}(\theta)\}.$$

Thus,

$$\text{lib}(\sigma) = (\text{lib}(\text{pc}[t : v_{\text{mgc}}^t], \text{lib}(\theta)) = (\text{lib}(\text{pc}[t : v_{\text{mgc}}^t], \text{lib}(\theta')) = \text{lib}(\sigma').$$

From this and $\text{lib}(\tau_1 \varphi) = \text{lib}(\tau_1)$, it follows that

$$\text{lib}(\sigma_0) \xrightarrow{\text{MGC}_n(A)}^* \text{lib}(\sigma').$$

- $\varphi \in \text{LibAct}$. This case is treated similarly to the previous one. We establish that $\text{lib}(\sigma) \xrightarrow{\text{MGC}_n(A)} \text{lib}(\sigma')$ and $\text{client}(\sigma') = \text{client}(\sigma)$. Since $\text{lib}(\tau_1 \varphi) = \text{lib}(\tau_1) \varphi$ and $\text{client}(\tau_1 \varphi) = \text{client}(\tau_1)$, which implies our claim.
- $\varphi \in \text{CallAct}$. In this case, there exist a thread $t \in \{1, \dots, n\}$, client nodes $v, v' \in \text{CNode}$, a method m , and a state $\theta \in \text{State}$, such that $(v, m(E), v')$ is in the control-flow relation of $C(A)$ and

$$\sigma = (\text{pc}[t : v], \theta) \wedge \sigma' = (\text{pc}[t : \langle \llbracket E \rrbracket \text{fst}(\theta) \rangle, v', \text{start}_m], \theta).$$

Then,

$$\begin{aligned}\text{client}(\sigma) &= (\text{client}(\text{pc})[t : v], \text{client}(\theta)) \wedge \\ \text{client}(\sigma') &= (\text{client}(\text{pc})[t : \langle v', m \rangle], \text{client}(\theta)) \wedge \\ \text{lib}(\sigma) &= (\text{lib}(\text{pc})[t : v_{\text{mgc}}^t], \text{lib}(\theta)) \wedge \\ \text{lib}(\sigma') &= (\text{lib}(\text{pc})[t : \langle \llbracket E \rrbracket(\text{fst}(\theta)), v_{\text{mgc}}^t, \text{start}_m \rangle], \text{lib}(\theta)).\end{aligned}$$

From this we can establish $\text{client}(\sigma) \xrightarrow{\varphi}_{C(\cdot)} \text{client}(\sigma')$ and $\text{lib}(\sigma) \xrightarrow{\varphi}_{\text{MGC}_n(A)} \text{lib}(\sigma')$. Furthermore, $\text{client}(\tau_1\varphi) = \text{client}(\tau_1)\varphi$ and $\text{lib}(\tau_1\varphi) = \text{lib}(\tau_1)\varphi$. Hence,

$$\text{client}(\sigma_0) \xrightarrow{\text{client}(\tau_1\varphi)^*}_{C(\cdot)} \text{client}(\sigma) \wedge \text{lib}(\sigma_0) \xrightarrow{\text{lib}(\tau_1\varphi)^*}_{\text{MGC}_n(A)} \text{lib}(\sigma).$$

- $\varphi \in \text{RetAct}$. This case is treated similarly to the previous one. We establish $\text{client}(\sigma) \xrightarrow{\varphi}_{C(\cdot)} \text{client}(\sigma')$ and $\text{lib}(\sigma) \xrightarrow{\varphi}_{\text{MGC}_n(A)} \text{lib}(\sigma')$. Just like in the previous case, we also have that $\text{client}(\tau_1\varphi) = \text{client}(\tau_1)\varphi$ and $\text{lib}(\tau_1\varphi) = \text{lib}(\tau_1)\varphi$, which implies our claim.

We have just shown that the claim holds for all φ .

Case 2. Assume $\eta \in \llbracket C(\cdot) \rrbracket$ and $\xi \in \llbracket A \rrbracket$ such that $\text{corehistory}(\eta) = \text{corehistory}(\xi)$. There exists a η -labelled client-local computation sequence that starts from an initial configuration $\sigma_0^1 \in \text{CConfig}$ and follows the transition relation $\longrightarrow_{C(\cdot)}$. Let n be the number of threads in $C(A)$. Since $\text{corehistory}(\eta) = \text{corehistory}(\xi)$, the thread identifiers mentioned in ξ all occur in η . Hence, using $\text{MGC}_n(A)$, we can generate a library-local computation sequence for ξ . This sequence is labelled with ξ , starts from some initial configuration $\sigma_0^2 \in \text{LConfig}$, and uses the transition relation $\longrightarrow_{\text{MGC}_n(A)}$. Out of these two computation sequences, we now construct the required trace $\tau \in \llbracket C(A) \rrbracket$ together with its computation sequence using $\longrightarrow_{C(A)}$. We first build a series of finite traces

$$\tau_0, \tau_1, \tau_2, \dots$$

and their computation sequences. One important feature of this series is that for $i < j$, the computation sequence 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 feature, this series has the limit computation sequence and the limit trace, which are the desired ones.

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

The client projection $\text{client}(\tau_i)$ is a finite prefix of η , and the library projection $\text{lib}(\tau_i)$ is a finite prefix of ξ . Furthermore, for some client configuration $\sigma_i^1 \in \text{CConfig}$, and some library configuration $\sigma_i^2 \in \text{LConfig}$, the constructed computation sequence for τ_i leads to $\sigma_i^1 \circ \sigma_i^2$:

$$\sigma_0^1 \circ \sigma_0^2 \xrightarrow{\tau_i^*}_{C(A)} \sigma_i^1 \circ \sigma_i^2,$$

and the given computation sequences for η and ξ have the following prefixes:

$$\sigma_0^1 \xrightarrow{\text{client}(\tau_i)^*}_{C(\cdot)} \sigma_i^1 \quad \wedge \quad \sigma_0^2 \xrightarrow{\text{lib}(\tau_i)^*}_{\text{MGC}_n(A)} \sigma_i^2.$$

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

Assume $\eta = \text{client}(\tau_i)\varphi_1\eta'$ and $\xi = \text{lib}(\tau_i)\varphi_2\xi'$ for some actions φ_1 and φ_2 and traces η' and ξ' (the case that $\eta = \text{client}(\tau_i)$ or $\xi = \text{lib}(\tau_i)$ is treated analogously). Let

the following be the transitions by φ_1 and φ_2 in the computation sequences for η and ξ :

$$\sigma_i^1 \xrightarrow{\varphi_1}_{C(\cdot)} \sigma^1 \quad \wedge \quad \sigma_i^2 \xrightarrow{\varphi_2}_{\text{MGC}_n(A)} \sigma^2$$

for some $\sigma^1 \in \text{CConfig}$ and $\sigma^2 \in \text{LConfig}$. We have the following cases:

- $\varphi_1, \varphi_2 \in \text{CallRetAct}$. In this case, φ_1 is the same as φ_2 . This is because $\text{corehistory}(\eta_1)$ and $\text{corehistory}(\xi_1)$ are the same by assumption so that their same-length prefixes $\text{corehistory}(\tau_i)\varphi_1$ and $\text{corehistory}(\tau_i)\varphi_2$ should be identical. Assume $\varphi_1 \in \text{CallAct}$ (the case when $\varphi_1 \in \text{RetAct}$ can be treated analogously). Then, there exist $t \in \{1, \dots, n\}$, $v, v' \in \text{CNode}$, $\theta^1 \in \text{CState}$, and $\theta^2 \in \text{LState}$ such that $(v, m(E), v')$ is in the control-flow relation of $C(A)$ and

$$\begin{aligned} \sigma_i^1 &= (\text{pc}[t : v], \theta^1) \quad \wedge \quad \sigma^1 = (\text{pc}[t : \langle v', m \rangle], \theta^1) \quad \wedge \\ \sigma_i^2 &= (\text{pc}[t : v_{\text{mgc}}^t], \theta^2) \quad \wedge \quad \sigma^2 = (\text{pc}[t : \langle \llbracket E \rrbracket(\text{fst}(\theta^1)), v_{\text{mgc}}^t, \text{start}_m \rangle], \theta^2). \end{aligned}$$

Hence,

$$\sigma_i^1 \circ \sigma_i^2 = (\text{pc}[t : v], \theta^1 \circ \theta^2) \quad \wedge \quad \sigma^1 \circ \sigma^2 = (\text{pc}[t : \langle \llbracket E \rrbracket(\text{fst}(\theta^1)), v', \text{start}_m \rangle], \theta^1 \circ \theta^2).$$

Furthermore, $\llbracket E \rrbracket(\text{fst}(\theta^1)) = \llbracket E \rrbracket(\text{fst}(\theta^1 \circ \theta^2))$. Then, $\sigma_i^1 \circ \sigma_i^2 \xrightarrow{\varphi_1}_{C(A)} \sigma^1 \circ \sigma^2$. Hence, in this case the desired τ_{i+1} is $\tau_i\varphi_1$, and the computation sequence for τ_{i+1} is:

$$\sigma_0^1 \circ \sigma_0^2 \xrightarrow{\tau_i}_{C(A)}^* \sigma_i^1 \circ \sigma_i^2 \xrightarrow{\varphi_1}_{C(A)} \sigma^1 \circ \sigma^2.$$

Finally, since $\text{client}(\tau_i\varphi_1) = \text{client}(\tau_i)\varphi_1$ and $\text{lib}(\tau_i\varphi_2) = \text{lib}(\tau_i)\varphi_2$, the conditions on the prefixes of the computation sequences of η and ξ in this case follow from those for τ_i .

- $\varphi_1 \in \text{RetAct}$, $\varphi_2 \in \text{LibAct}$ and they are performed by the same thread. In this case for some $t \in \{1, \dots, n\}$, $\theta^1 \in \text{CState}$, $\theta^2 \in \text{LState}$, $\theta^3 \in \text{LState}$, $v_0 \in \text{CNode}$, $v, v' \in \text{LNode}$ and $k \in \text{Val}$, we have that (v, c, v') is in the control-flow relation of $C(A)$ and

$$\begin{aligned} \sigma_i^1 &= (\text{pc}[t : \langle v_0, \text{method}(v) \rangle], \theta^1) \quad \wedge \\ \sigma_i^2 &= (\text{pc}[t : \langle k, v_{\text{mgc}}^t, v \rangle], \theta^2) \quad \wedge \\ \sigma^2 &= (\text{pc}[t : \langle \theta^3(\text{param}), v_{\text{mgc}}^t, v' \rangle], \theta^3) \quad \wedge \quad \theta^3 \in f_c(\theta^2[\text{param} : k]). \end{aligned}$$

Then

$$\begin{aligned} \sigma_i^1 \circ \sigma_i^2 &= (\text{pc}[t : \langle k, v_0, v \rangle], \theta^1 \circ \theta^2) \quad \wedge \\ \sigma_i^1 \circ \sigma^2 &= (\text{pc}[t : \langle (\theta^1 \circ \theta^3)(\text{param}), v_0, v' \rangle], \theta^1 \circ \theta^3). \end{aligned}$$

From (1) in § 2, we have

$$\begin{aligned} \theta^1 \circ \theta^3 &\in \{\theta^1\} \circ f_c(\theta^2[\text{param} : k]) = f_c(\theta^1 \circ (\theta^2[\text{param} : k])) \\ &= f_c((\theta^1 \circ \theta^2)[\text{param} : k]). \end{aligned}$$

Hence, $\sigma_i^1 \circ \sigma_i^2 \xrightarrow{\varphi_2}_{C(A)} \sigma_i^1 \circ \sigma^2$. The desired τ_{i+1} is $\tau_i\varphi_2$, and its computation sequence is:

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

Finally, the conditions on the prefixes of the computation sequences for η and ξ follow from $\text{client}(\tau_i\varphi_2) = \text{client}(\tau_i)$ and $\text{lib}(\tau_i\varphi_2) = \text{lib}(\tau_i)\varphi_2$.

- $\varphi_1 \in \text{ClientAct}$, $\varphi_2 \in \text{CallAct}$ and they are performed by the same thread. This case is treated similarly to the previous one: we show that $\sigma_i^1 \circ \sigma_i^2 \xrightarrow{\varphi_1}_{C(A)} \sigma^1 \circ \sigma_i^2$.
- $\varphi_1 \in \text{ClientAct}$, $\varphi_2 \in \text{LibAct}$ and they are done by different threads. In this case,

$$\text{client}(\tau_i\varphi_1\varphi_2) = \text{client}(\tau_i)\varphi_1 \quad \wedge \quad \text{lib}(\tau_i\varphi_1\varphi_2) = \text{lib}(\tau_i)\varphi_2.$$

As before, we can show that

$$\sigma_i^1 \circ \sigma_i^2 \xrightarrow{C(A)} \sigma_i^1 \circ \sigma_i^2 \quad \wedge \quad \sigma_i^1 \circ \sigma_i^2 \xrightarrow{C(A)} \sigma_i^1 \circ \sigma_i^2.$$

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

$$\sigma_0^1 \circ \sigma_0^2 \xrightarrow{C(A)} \sigma_0^1 \circ \sigma_0^2.$$

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

- $\varphi_1 \in \text{CallRetAct}$, $\varphi_2 \in \text{LibAct}$ and they are performed by different threads. In this case

$$\text{client}(\tau_i \varphi_2) = \text{client}(\tau_i) \quad \wedge \quad \text{lib}(\tau_i \varphi_2) = \text{lib}(\tau_i) \varphi_2.$$

Here we can show that

$$\sigma_0^1 \circ \sigma_0^2 \xrightarrow{C(A)} \sigma_i^1 \circ \sigma_i^2 \xrightarrow{C(A)} \sigma_i^1 \circ \sigma_i^2,$$

which implies that $\tau_i \varphi_2$ is the desired trace.

- $\varphi_1 \in \text{ClientAct}$, $\varphi_2 \in \text{CallRetAct}$ and they are performed by different threads. This case is similar to the previous one.
- φ_1 and φ_2 are done by the same thread and one of the two conditions holds: $\varphi_1 \in \text{ClientAct}$ and $\varphi_2 \in (\text{RetAct} \cup \text{LibAct})$, or $\varphi_1 \in \text{CallAct}$ and $\varphi_2 \in \text{LibAct}$. Using the definition of \circ , it is easy to check that these cases are impossible.

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

B.2 Proof of Lemma 11

Below we sometimes write \sqsubseteq_π and \sim_π instead of \sqsubseteq and \sim to make the bijection π used to establish the relations between histories or traces explicit. For actions φ and ψ in a history H we write $\varphi \prec_H \psi$ if φ precedes ψ in H .

Consider a well-formed client trace $\tau \in \text{CTrace}$ and well-formed core histories $H, S \in \text{CoreHistory}$ such that $\text{corehistory}(\tau) = H$ and $H \sqsubseteq S$. We need to prove that there exists a well-formed client trace $\tau' \in \text{CTrace}$ such that $\text{corehistory}(\tau') = S$ and $\tau \sim \tau'$. To this end, we define a (possibly infinite) sequence of steps that transforms τ into τ' . The trace τ' is constructed as a limit of a sequence of well-formed traces ξ_k , defined for every finite prefix S_k of S of length k . This sequence is defined inductively, and every trace ξ_k is such that for some prefix κ_k of ξ_k we have $\text{corehistory}(\kappa_k) = S_k$ and $\text{corehistory}(\xi_k) \sqsubseteq_\pi S$, where π is an identity on actions from S_k in $\text{corehistory}(\xi_k)$. To construct the sequence, we let $\xi_0 = \tau$ and let the prefix κ_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 $\tau_1 = \kappa_k$, $\tau_1 \tau_2 = \xi_k$, $H_1 = S_k$ and $H_1 \psi H_2 = S$.

Lemma 13. *Consider a well-formed core history $H_1 \psi H_2$, where $\psi \in \text{CallRetAct}$, and a well-formed client trace $\tau_1 \tau_2$ such that*

$$\text{corehistory}(\tau_1) = H_1, \tag{2}$$

$$\text{corehistory}(\tau_1 \tau_2) \sqsubseteq_\pi H_1 \psi H_2, \tag{3}$$

where π is an identity on actions from H_1 in $\text{corehistory}(\tau_1 \tau_2)$. Then there exist traces τ_2' and τ_2'' such that $\tau_1 \tau_2' \tau_2''$ is a well-formed trace and

$$\tau_1 \tau_2 \sim_\rho \tau_1 \tau_2' \tau_2'', \tag{4}$$

$$\text{corehistory}(\tau_1 \tau_2') = H_1 \psi, \tag{5}$$

$$\text{corehistory}(\tau_1 \tau_2' \tau_2'') \sqsubseteq_{\pi'} H_1 \psi H_2, \tag{6}$$

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

It is easy to see that for all i, j with $i < j$ and for all prefixes κ_i and κ_j of the traces ξ_i and ξ_j thus constructed, κ_i is a prefix of κ_j . Hence, the sequence of traces κ_k has a limit trace τ' such that for every k , κ_k is a prefix of τ' and $\text{corehistory}(\tau') = S$, which shows that Lemma 13 indeed entails Lemma 11.

To prove Lemma 13, we transform $\tau_1\tau_2$ into $\tau_1\tau_2'\tau_2''$ by applying a finite number of the following transformations that preserve its properties of interest, described by the proposition below.

Proposition 14. *Let τ be a well-formed client trace and H a well-formed core history such that $\text{corehistory}(\tau) \sqsubseteq_\pi H$. Then swapping any two adjacent actions φ_1 and φ_2 in τ executed by different threads such that*

1. $\varphi_1 \in \text{CallAct}$ and $\varphi_2 \notin \text{RetAct}$; or
2. $\varphi_2 \in \text{RetAct}$ and if $\varphi_1 \in \text{CallAct}$, then $\varphi_2 \prec_H \varphi_1$,

yields a well-formed trace τ' such that $\tau \sim_\rho \tau'$ and $\text{corehistory}(\tau') \sqsubseteq_{\pi'} H$.

The bijection π' is defined as follows. If $\varphi_1 \in \text{ClientAct}$ or $\varphi_2 \in \text{ClientAct}$, then $\pi' = \pi$. Otherwise, let i be the index of φ_1 in $\text{corehistory}(\tau)$. 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 $k \notin \{i, i+1\}$.

Proof of Lemma 13. From (2) and (3) it follows that $\tau_2 = \tau_3'\psi\tau_4'$ for some traces τ_3' and τ_4' . We have two cases.

1. $\psi \in \text{CallAct}$. Then τ_3' cannot contain a return action φ , because in this case we would have $\varphi \prec_{\text{corehistory}(\tau_1\tau_2)} \psi$, but $\varphi \in H_2$ and, thus, $\varphi \not\prec_{H_1\psi H_2} \psi$, which contradicts (3). Hence, there are no return actions in τ_3' . Moreover, since $\tau_1\tau_2$ is well-formed, for any call action $\varphi = (t, \text{call } m(k))$ in τ_3' there are no actions by the thread t in τ_3' following φ . Thus, we can move all call actions 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 14(1). We thus obtain the trace $\tau_1\tau_3''\psi\tau_4''\tau_4'$, where τ_4'' consists of call actions in τ_3' and τ_3'' of the rest of actions in the subtrace. Conditions (4)–(6) then follow from Proposition 14(1) and the transitivity of \sim for $\tau_2' = \tau_3''\psi$ and $\tau_2'' = \tau_4''\tau_4'$.

2. $\psi \in \text{RetAct}$. Since the history $\text{corehistory}(\tau_1\tau_2)$ is well-formed, the matching call of ψ is in H_1 . From (2) and (3) 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 14(2). We thus obtain the trace $\tau_1\psi\tau_3'\tau_4'$. Conditions (4)–(6) then follow from Proposition 14(2) and the transitivity of \sim for $\tau_2' = \psi$, $\tau_2'' = \tau_3'\tau_4'$.

B.3 Proof of Lemma 12

Let n be the number of threads in $C(\cdot)$. For a state $\theta \in \text{CState}$, let $\text{core}(\theta)$ be the state θ with the stack excluding the retval_t variables. (Strictly speaking, we are overloading the word “state” here, because $\text{core}(\theta)$ has the type $(\text{CVar}' \rightarrow \text{Val}) \times \text{Heap}$ for $\text{CVar}' = \text{CVar} - \{\text{retval}_t \mid t = 1..n\}$.) We lift core to sets of states pointwise and let $\text{core}(\text{pc}, \theta) = \text{core}(\theta)$. For $\sigma \in \text{CConfig}$, $t \in \{1, \dots, n\}$ and variable $x \in \text{CVar}$, we write $\sigma(t)$ for $(\text{fst}(\sigma))(t)$ (i.e., the thread position of t in σ), and $\sigma(x)$ for $((\text{fst} \circ \text{snd})(\sigma))(x)$ (i.e., the value of the variable x in σ).

Take $\tau \in \llbracket C(\cdot) \rrbracket$ and η such that $\tau \sim \eta$. Let us fix the bijection π used to establish this equivalence. We consider only the case that τ is infinite (the finite case is easier and can be handled using the same ideas). Since $\tau \sim \eta$, we have

$$\text{visible}(\tau) = \text{visible}(\eta).$$

Furthermore, there exists a τ -labelled computation sequence from some initial configuration $\sigma_0 \in \text{CConfig}$ such that each step of the sequence uses the relation $\longrightarrow_{C(\cdot)}$. That is, if $\tau = \varphi_1 \varphi_2 \varphi_3 \dots$, we have

$$\sigma_0 \xrightarrow{\varphi_1}_{C(\cdot)} \sigma_1 \xrightarrow{\varphi_2}_{C(\cdot)} \sigma_2 \xrightarrow{\varphi_3}_{C(\cdot)} \sigma_3 \xrightarrow{\varphi_4}_{C(\cdot)} \dots \quad (7)$$

for some $\sigma_i \in \text{CConfig}$. Using this computation sequence for τ , we construct a series of finite computation sequences and define the desired computation sequence for η as the limit of this series.

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

1. The i -th computation sequence has the form

$$\sigma'_0 \xrightarrow{\psi_1}_{C(\cdot)} \sigma'_1 \xrightarrow{\psi_2}_{C(\cdot)} \sigma'_2 \xrightarrow{\psi_3}_{C(\cdot)} \dots \xrightarrow{\psi_i}_{C(\cdot)} \sigma'_i$$

where $\sigma'_0 = \sigma_0$, the trace $\psi_1 \dots \psi_i$ is a prefix of η and $\sigma'_1, \dots, \sigma'_i \in \text{CConfig}$.

2. For all $j \leq i$, if τ_1 is a finite prefix of τ such that $\text{visible}(\tau_1) = \text{visible}(\psi_1 \dots \psi_j)$, then

$$\text{core}(\sigma_{|\tau_1|}) = \text{core}(\sigma'_j).$$

3. For all $j \leq i$ and all $t \in \{1, \dots, n\}$, if τ_1 is a finite prefix of τ such that $\tau_1|_t = (\psi_1 \dots \psi_j)|_t$, then

$$\sigma_{|\tau_1|}(t) = \sigma'_j(t) \quad \wedge \quad \sigma_{|\tau_1|}(\text{retval}_t) = \sigma'_j(\text{retval}_t).$$

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

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

$$\sigma'_i \xrightarrow{\psi_{i+1}}_{C(\cdot)} \sigma'_{i+1}$$

and σ'_{i+1} satisfies the second and third assumptions above. We show that such a σ'_{i+1} exists by a case analysis on ψ_{i+1} .

- $\psi_{i+1} = (t, \text{Client}(c))$ for some $t \in \{1, \dots, n\}$ and $c \in \text{PComm}$. Let φ_k be the action of τ that is mapped to ψ_{i+1} by π . Then, the computation sequence of τ in (7) gives us the transition:

$$\sigma_{k-1} \xrightarrow{\varphi_k}_{C(\cdot)} \sigma_k \quad (8)$$

Let $\sigma_k = (\text{pc}_k, (s_k, h_k))$, and let $\sigma'_i = (\text{pc}'_i, (s'_i, h'_i))$. We define s'_{i+1} and σ'_{i+1} as follows:

$$s'_{i+1}(x) = \begin{cases} s'_i(x), & \text{if } x \text{ is } \text{retval}_{t'} \text{ for } t' \neq t; \\ s_k(x), & \text{otherwise;} \end{cases}$$

$$\sigma'_{i+1} = (\text{pc}_i[t : \text{pc}_k(t)], (s'_{i+1}, h_k)).$$

We show that σ'_{i+1} is the desired configuration. Since π preserves the order of actions from ClientAct , $\text{visible}(\varphi_1 \dots \varphi_{k-1}) = \text{visible}(\psi_1 \dots \psi_i)$. Then assumption 2 for the i -th computation sequence implies

$$\text{core}(\sigma_{k-1}) = \text{core}(\sigma'_i).$$

Furthermore, since π preserves the order of actions within the same thread, $(\varphi_1 \dots \varphi_{k-1})|_t = (\psi_1 \dots \psi_i)|_t$. Then assumption 3 for the i -th computation sequence implies

$$\sigma_{k-1}(t) = \sigma'_i(t) \quad \wedge \quad \sigma_{k-1}(\text{retval}_t) = \sigma'_i(\text{retval}_t).$$

Recall that the transitions for thread t do not access the positions of other threads or $\text{retval}_{t'}$ for $t' \neq t$. Additionally, $\varphi_k = \psi_{i+1}$. Hence, transition (8) and the definition of σ'_{i+1} entails that the following transition is also possible:

$$\sigma'_i \xrightarrow{\psi_{i+1}}_{C(\cdot)} \sigma'_{i+1}.$$

Now it remains to show that σ'_{i+1} satisfies the conditions in assumptions 2 and 3. To handle assumption 2, consider a finite prefix τ_1 of τ such that $\text{visible}(\tau_1) = \text{visible}(\psi_1 \dots \psi_{i+1})$. Then

$$\text{core}(\sigma_{|\tau_1|}) = \text{core}(\sigma_k).$$

But $\text{core}(\sigma_k) = \text{core}(\sigma'_{i+1})$ by the definition of σ'_{i+1} . Hence $\text{core}(\sigma_{|\tau_1|}) = \text{core}(\sigma'_{i+1})$, as desired. For assumption 3, we take an arbitrary thread $t' \in \{1, \dots, n\}$ and consider a finite prefix τ_1 of τ such that $\tau_1|_{t'} = (\psi_1 \dots \psi_{i+1})|_{t'}$. If $t' \neq t$, we have

$$\tau_1|_{t'} = (\psi_1 \dots \psi_i \psi_{i+1})|_{t'} = (\psi_1 \dots \psi_i)|_{t'}.$$

Hence, by assumption 3 on σ'_i ,

$$\sigma_{|\tau_1|}(t') = \sigma'_i(t') \quad \wedge \quad \sigma_{|\tau_1|}(\text{retval}_{t'}) = \sigma'_i(\text{retval}_{t'}).$$

On the other hand, by the definition of σ'_{i+1} ,

$$\sigma'_i(t') = \sigma'_{i+1}(t') \quad \wedge \quad \sigma'_i(\text{retval}_{t'}) = \sigma'_{i+1}(\text{retval}_{t'}).$$

From the four equalities above, the condition in assumption 3 follows. If $t' = t$,

$$\sigma_{|\tau_1|}(t') = \sigma_k(t') = \sigma'_{i+1}(t') \quad \wedge \quad \sigma_{|\tau_1|}(\text{retval}_{t'}) = \sigma_k(\text{retval}_{t'}) = \sigma'_{i+1}(\text{retval}_{t'}),$$

which again implies the condition in assumption 3.

- $\varphi \in (t, \text{call } m(k'))$ for some $t \in \{1, \dots, n\}$ and $k' \in \text{Val}$. Let φ_k be the action of τ that is mapped to ψ_{i+1} by π . Then, the computation sequence of τ in (7) gives us the transition:

$$\sigma_{k-1} \xrightarrow{\varphi_k}_{C(\cdot)} \sigma_k \tag{9}$$

Let $m(E)$ be the instruction executed by this transition. Then,

$$\llbracket E \rrbracket((\text{fst} \circ \text{snd})(\sigma_{k-1})) = u. \tag{10}$$

Now, define σ'_{i+1} as identical to σ'_i except $\sigma'_{i+1}(t) = \sigma_k(t)$. We show that σ'_{i+1} is the configuration we are looking for. Since π preserves the order of actions within threads, $(\varphi_1 \dots \varphi_k)|_t = (\psi_1 \dots \psi_{i+1})|_t$. As φ_k and ψ_{i+1} are the same action by thread t , this implies

$$(\varphi_1 \dots \varphi_{k-1})|_t = (\psi_1 \dots \psi_i)|_t.$$

Hence, assumption 3 for the i -th computation sequence gives us

$$\sigma_{k-1}(t) = \sigma'_i(t) \quad \wedge \quad \sigma_{k-1}(\text{retval}_t) = \sigma'_i(\text{retval}_t). \tag{11}$$

We will argue that for all variables x local to thread t ,

$$\sigma_{k-1}(x) = \sigma'_i(x). \tag{12}$$

Note that (11) already takes care of the case that x is the variable retval_t . We prove the remaining cases by doing the case analysis on whether or not $(\psi_1 \dots \psi_i)|_t$ contains any actions in ClientAct . First, consider the “not contain” case. In this case,

$(\varphi_1 \dots \varphi_{k-1})|_t$ does not contain ClientAct-type actions, either. Since only actions in ClientAct by thread t can change variables x local to thread t with $x \neq \text{retval}_t$, we have that

$$\sigma_{k-1}(x) = \sigma_0(x) = \sigma'_i(x) \quad \text{for all such variables } x.$$

Hence, (12) holds in this case. Next, consider the ‘‘contain’’ case. Let ψ_j be the last ClientAct-type action in $(\psi_1 \dots \psi_i)|_t$, and φ_l the corresponding action in $(\varphi_1 \dots \varphi_{k-1})|_t$. In this case,

$$\text{visible}(\varphi_1 \dots \varphi_l) = \text{visible}(\psi_1 \dots \psi_j).$$

So, by assumption 2 on σ'_j , we have that

$$\text{core}(\sigma_l) = \text{core}(\sigma'_j).$$

Furthermore, by the choice of ψ_j and φ_l , if x is a variable local to thread t and it is not retval_t , then

$$\sigma_{k-1}(x) = \sigma_l(x) \quad \wedge \quad \sigma'_i(x) = \sigma'_j(x).$$

From the three equations above follows the equation in (12). This completes our proof of (12).

It is now time to use (12). Recall that the parameter expression E can include only those variables local to the thread t . Furthermore, because of (12), all t -local variables have the same values in σ_{k-1} and σ'_i . Hence,

$$\llbracket E \rrbracket((\text{fst} \circ \text{snd})(\sigma_{k-1})) = \llbracket E \rrbracket((\text{fst} \circ \text{snd})(\sigma'_i)). \quad (13)$$

From (9)-(13) and the definition of σ'_{i+1} , it follows that

$$\sigma'_i \xrightarrow{\psi_{i+1}}_{C(\cdot)} \sigma'_{i+1}.$$

It remains to show that σ'_{i+1} satisfies the requirements in assumptions 2 and 3. The requirement for assumption 2 holds for σ'_{i+1} , because

$$\text{visible}(\psi_1 \dots \psi_i) = \text{visible}(\psi_1 \dots \psi_{i+1}) \quad \wedge \quad \text{core}(\sigma'_i) = \text{core}(\sigma'_{i+1})$$

and σ'_i already satisfies the same requirement. Our proof for requirement 3 is the same as in the previous case.

- $\psi_{i+1} = (t, \text{ret } m(u))$ for some $t \in \{1, \dots, n\}$ and $u \in \text{Val}$. Let φ_k be the action in the trace τ that is related to ψ_{i+1} by the bijection π . Then, the computation sequence of τ in (7) gives us the transition:

$$\sigma_{k-1} \xrightarrow{\varphi_k}_{C(\cdot)} \sigma_k$$

Since π preserves the order of actions by the same thread, $(\varphi_1 \dots \varphi_{k-1})|_t = (\psi_1 \dots \psi_i)|_t$. Hence, by assumption 3 for the i -th computation sequence,

$$\sigma'_i(t) = \sigma_{k-1}(t).$$

Let σ'_{i+1} be the same as σ'_i except $\sigma'_{i+1}(\text{retval}_t) = \sigma_k(\text{retval}_t)$ and $\sigma'_{i+1}(t) = \sigma_k(t)$. Then, there exists a transition

$$\sigma'_i \xrightarrow{\psi_{i+1}}_{C(\cdot)} \sigma'_{i+1}.$$

Assumption 2 holds for σ'_{i+1} because $\text{core}(\sigma'_i) = \text{core}(\sigma'_{i+1})$, $\text{visible}(\psi_1 \dots \psi_i) = \text{visible}(\psi_1 \dots \psi_i \psi_{i+1})$ and σ'_i already satisfies assumption 2. The remaining assumption 3 follows from the definition of σ'_{i+1} and the fact that only the actions by thread t can change its thread position and the retval_t variable. □

C Proof of Theorem 9 and additional material for Section 6

Proof Theorem 9. By contrapositive, assume $\llbracket C(A_1) \rrbracket$ has an infinite trace τ . From Lemma 10(1), we have that $\text{lib}(\tau) \in \llbracket A_1 \rrbracket$. If $\text{client}(\tau)$ is finite, then the trace $\text{lib}(\tau)$ has infinitely many actions from LibAct , but only finitely many from CallRetAct . By the definition of $\llbracket A_1 \rrbracket$, this means that $\text{history}(\text{lib}(\tau))$ does not satisfy LF, which contradicts the lock-freedom of A_1 .

Assume now $\text{client}(\tau)$ is infinite. By Theorem 4, there exists a trace $\tau' \in \llbracket C(A_3) \rrbracket$ with infinitely many actions from $\text{ClientAct} \cup \text{CallRetAct}$. From this trace we can easily construct an infinite trace $\tau'' \in \llbracket C(A_2) \rrbracket$ if we replace any divergence at skip° in A_3 by suspending the corresponding thread forever. (Note that we can do this because the semantics in Figure 1 allows for unfair schedulers.) \square

Reduction from lock-freedom to termination. For a library $A = \{m = C_m \mid m \in M\}$ let $\llbracket A \rrbracket_1$ be defined as $\llbracket A \rrbracket$ (see §3), but where the CFG of thread t in $\text{MGC}_n(A)$ is $(N_t, T_t, v_{\text{mgc}}^t, \text{end}_{\text{mgc}}^t)$ with $N_t = \{v_{\text{mgc}}^t \mid t = 1..n\} \cup \{\text{end}_{\text{mgc}}^t \mid t = 1..n\}$ and $T_t = \{(v_{\text{mgc}}^t, m(k), \text{end}_{\text{mgc}}^t) \mid m \in M, k \in \text{Val}\}$. Thus, $\llbracket A \rrbracket_1$ defines the set of traces of an arbitrary number of library operations running in parallel. We can now formulate the reduction from lock-freedom to termination we described informally in §6.

Theorem 15 ([7]). *A library A is lock-free iff there are no infinite traces in $\llbracket A \rrbracket_1$.*

Counterexample to the compositionality of lock-freedom under fair scheduling.

We now provide a counterexample showing that lock-freedom under fair scheduling is not compositional. Consider the following library A_1 :

```
int x = 1;
void wait() { while (x == 0); }
void blip() { x = 0; x = 1; }
```

and another library B built on top of A_1 :

```
int y = 0;
void m1() { wait(); y = 1; }
void m2() { while (y == 0) blip(); }
```

The library A_1 is lock-free under fair scheduling: in a program using it some method always terminates. This may not be true under an unfair one, which can suspend a `blip` operation forever in between `x = 0` and `x = 1`. The library A_1 is also linearizable with respect to the specification

$$A_3 = \{\text{wait} = \text{blip} = (\text{skip}^\circ; \text{skip}; \text{skip}^\circ)\}.$$

The library B is lock-free assuming that it uses

$$A_2 = \{\text{wait} = \text{blip} = \text{skip}\}.$$

However, the library B implemented in terms of A_1 is not lock-free: `m2` can continually execute `blip`, which will prevent `wait` in `m1` from returning. The statement `y = 1` in `m1`, which could stop `blips`, will thus never be executed.