Liveness-Preserving Atomicity Abstraction*

Alexey Gotsman and Hongseok Yang

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

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 its 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

* We would like to thank Anindya Banerjee, Aleks Nanevski, Matthew Parkinson and Viktor Vafeiadis for helpful comments and suggestions. Yang was supported by EPSRC.

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, both for safety and 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, 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 that lock-freedom is a compositional property for linearizable libraries (Theorem 9, §6): 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).

For space reasons, proofs are omitted in the paper. They can be found in [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 $x, y, \ldots$, 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:

$$
\begin{array}{llll}
E & ::= \mathbb{Z} \mid x \mid E + E \mid -E \mid \ldots & C & ::= \; c \mid m(E) \mid C; C \mid C + C \mid C^{\circlearrowleft} \\
D & ::= C; \mathsf{return}(E) & \multicolumn{2}{l}{(\text{where } C \text{ does not include method calls } m(E))} \\
A & ::= \{m = D, \ldots, m = D\} & & \\
C(A) & ::= \mathsf{let}\ A\ \mathsf{in}\ C \parallel \ldots \parallel C & \multicolumn{2}{l}{(\text{where all the methods called are defined in } A)}
\end{array}
$$

The language includes primitive commands c, method call $m(E)$, sequential composition $C; C'$, nondeterministic choice $C{+}C'$, and iteration $C^{\circlearrowleft}$. We use a notation $C^{\circlearrowleft}$ here (instead of the usual Kleene star C^*), so as to emphasise that $C^{\circlearrowleft}$ 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 $\mathsf{assume}(E)$, 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 [7, §A]).

Let us fix a program $C(A) = \mathsf{let}\ A\ \mathsf{in}\ C_1 \parallel \ldots \parallel C_n$ 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: $\mathsf{sig}(A) = M$. In the following we index threads in programs using the set of identifiers $\mathsf{ThreadID} = \mathbb{N}$.

We restrict programs to ensure that the state of the client is disjoint from that of the library, and that 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]; we intend to relax it in our future work. Technically, we assume $\mathsf{PComm} = \mathsf{ClientPComm} \uplus \mathsf{LibPComm}$ 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 $\mathsf{retval}_t \in \mathsf{Var}$ for each thread t and $\mathsf{param} \in \mathsf{Var}$. The variable retval_t contains the result of the most recent method call by thread t 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 keeps the values of parameters passed upon method calls and is implicitly updated by the call command. We assume that 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 $\mathsf{Var} = \mathsf{CVar} \uplus \mathsf{LVar}$ for some sets CVar and LVar representing variables that belong to the client and the library, respectively. Let $\mathsf{param} \in \mathsf{LVar}$ and $\mathsf{retval}_t \in \mathsf{CVar}$, $t = 1..n$. We then define the set of program states State as follows:

$$\mathsf{Loc} = \mathsf{CLoc} \uplus \mathsf{LLoc} \qquad \mathsf{Val} = \mathbb{Z} \uplus \mathsf{Loc}$$
$$\mathsf{Stack} = \mathsf{Var} \to \mathsf{Val} \qquad \mathsf{Heap} = \mathsf{Loc} \to \mathsf{Val} \qquad \mathsf{State} = \mathsf{Stack} \times \mathsf{Heap}$$

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: $\mathsf{CState} = (\mathsf{CVar} \to \mathsf{Val}) \times (\mathsf{CLoc} \to \mathsf{Val})$ and $\mathsf{LState} = (\mathsf{LVar} \to \mathsf{Val}) \times (\mathsf{LLoc} \to \mathsf{Val})$. Also, we define three operations relating these sets to State: $\mathsf{client} : \mathsf{State} \to \mathsf{CState}$, $\mathsf{lib} : \mathsf{State} \to \mathsf{LState}$ and $\circ : \mathsf{CState} \times \mathsf{LState} \to \mathsf{State}$. The first two project states to client and library states by restricting the domains of the stack and the heap: e.g., for $(s, h) \in \mathsf{State}$ we have $\mathsf{lib}(s, h) = (s|_{\mathsf{LVar}}, h|_{\mathsf{LLoc}})$. 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 $\circ$ 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 $(N, T, \mathsf{start}, \mathsf{end})$, consisting of the set of program positions N, the control-flow relation $T \subseteq N \times \mathsf{Comm} \times N$, and the initial and final positions $\mathsf{start}, \mathsf{end} \in N$. The edges are annotated with commands from Comm, which are primitive commands, calls $m(E)$ or returns $\mathsf{return}(E)$. Every command C in our language can be translated to a CFG in a standard manner [7, §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 $(N_t, T_t, \mathsf{start}_t, \mathsf{end}_t)$, and

the body D_m of a method m by $(N_m, T_m, \mathsf{start}_m, \mathsf{end}_m)$. We often view this collection of CFGs for $C(A)$ as a single graph consisting of two node sets $\mathsf{CNode} = \biguplus_{t=1}^{n} N_t$ and $\mathsf{LNode} = \biguplus_{m \in \mathsf{sig}(A)} N_m$, and the edge set $T = \biguplus_{t=1}^{n} T_t \uplus \biguplus_{m \in \mathsf{sig}(A)} T_m$. Finally, we define $\mathsf{method} : \mathsf{LNode} \to M$ as follows: $\mathsf{method}(v) = m$ 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

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

where $t \in \mathsf{ThreadID}$, $c \in \mathsf{PComm}$, $k \in \mathsf{Val}$ and $m \in \mathsf{Method}$. Each action corresponds 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 let $\mathsf{CallRetAct} = \mathsf{CallAct} \cup \mathsf{RetAct}$. 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 as a transformer f_c of type $\mathsf{LState} \to \mathcal{P}(\mathsf{LState})$, if $c \in \mathsf{LibPComm}$, or of type $\mathsf{CState} \to \mathcal{P}(\mathsf{CState})$, if $c \in \mathsf{ClientPComm}$. In both cases, we lift it to a function $f_c : \mathsf{State} \to \mathcal{P}(\mathsf{State})$ by transforming only the client or the library part of the input state:

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

For the transformers of sample primitive commands see [7, §A].

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

The semantics of the program $C(A)$ with threads $\{1, \ldots, n\}$ is defined using a transition relation $\longrightarrow_{C(A)} : \mathsf{Config} \times \mathsf{Act} \times \mathsf{Config}$, which transforms program configurations $\mathsf{Config} = (\{1, \ldots, n\} \to \mathsf{Pos}) \times \mathsf{State}$. 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 variable for the thread t executing the return command.

Our operational semantics induces a trace interpretation of programs $C(A)$. For a finite trace τ and $\sigma, \sigma' \in \mathsf{Config}$ 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 $\sigma \in \mathsf{Config}$ we write $\sigma \xrightarrow{\tau}{}^{\omega}_{C(A)} -$ 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 $[1 : \mathsf{start}_1, \ldots, n : \mathsf{start}_n]$. The trace semantics of $C(A)$ is defined as follows:

$$[\![C(A)]\!] = \{\tau \mid \exists \theta \in \mathsf{State}.\ (\mathsf{pc}_0, \theta) \xrightarrow{\tau}{}^{\omega}_{C(A)} - \ \vee \ \exists \sigma \in \mathsf{Config}.\ (\mathsf{pc}_0, \theta) \xrightarrow{\tau}{}^{*}_{C(A)} \sigma\}.$$

$$\frac{(v, c, v') \in T \quad \theta' \in f_c(\theta)}{\mathsf{pc}[t : v], \theta \xrightarrow{(t,\, \mathsf{Client}(c))}_{C(A)} \mathsf{pc}[t : v'], \theta'} \qquad \frac{(v, m(E), v') \in T \quad [\![E]\!]s = u}{\mathsf{pc}[t : v], \theta \xrightarrow{(t,\, \mathsf{call}\ m(u))}_{C(A)} \mathsf{pc}[t : \langle u, v', \mathsf{start}_m \rangle], \theta}$$

$$\frac{(v, c, v') \in T \quad (s', h') \in f_c(s[\mathsf{param} : u], h)}{\mathsf{pc}[t : \langle u, v_0, v \rangle], (s, h) \xrightarrow{(t,\, \mathsf{Lib}(c))}_{C(A)} \mathsf{pc}[t : \langle s'(\mathsf{param}), v_0, v' \rangle], (s', h')}$$

$$\frac{(v, \mathsf{return}(E), v') \in T \quad [\![E]\!](s[\mathsf{param} : u]) = u'}{\mathsf{pc}[t : \langle u, v_0, v \rangle], (s, h) \xrightarrow{(t,\, \mathsf{ret}\ (\mathsf{method}(v))(u'))}_{C(A)} \mathsf{pc}[t : v_0], (s[\mathsf{retval}_t : u'], h)}$$

Fig. 1. Operational semantics of programs. $[\![E]\!]s \in \mathsf{Val}$ 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 g everywhere except x, where it has the value y. We remind the reader that T is the control-flow relation of $C(A)$.

Note that $[\![C(A)]\!]$ 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 $\mathsf{CallRetAct} \cup \mathsf{ClientAct}$, and ***library traces***, which include only actions from $\mathsf{CallRetAct} \cup \mathsf{LibAct}$. Given a trace $\tau \in \mathsf{Trace}$, we can obtain the corresponding client $\mathsf{client}(\tau)$ and library $\mathsf{lib}(\tau)$ traces by projecting τ to the appropriate subsets of actions. We consider two further projections: $\mathsf{visible}(\tau)$ projects τ to actions in $\mathsf{ClientAct}$, 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 $\mathsf{CallRetAct} \cup \mathsf{BlockAct}$, where $\mathsf{BlockAct} = \{(t, \mathsf{starve}) \mid t \in \mathsf{ThreadID}\}$. An action (t, starve) means that thread t is suspended by the scheduler and is never scheduled again (is 'starved'). We record starve events because the liveness properties we are dealing with in this paper, such as lock-freedom (§6), need to distinguish between method non-termination due to divergence and the one due to being starved by the scheduler. Let $\mathsf{History}$ be the set of all histories.

Given a trace $\tau \in \mathsf{Trace}$, we can construct a corresponding history $\mathsf{history}(\tau)$ in two steps. First, for every thread t, if the last action of the thread in τ exists and is an action in $\mathsf{CallAct} \cup \mathsf{LibAct}$, we insert (t, starve) right after this action in τ, obtaining a sequence of actions τ'. The history $\mathsf{history}(\tau)$ is then obtained by projecting τ' to actions in $\mathsf{CallRetAct} \cup \mathsf{BlockAct}$.

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 $\forall t \in \mathsf{ThreadID}.\, H|_t = H'|_t$ and there is a bijection $\pi \colon \{1, \ldots, |H|\} \to \{1, \ldots, |H'|\}$ such that $\forall i.\, H(i) = H'(\pi(i))$ and*

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

That is, the history H' linearizes the history H when it is a permutation of the latter preserving the order of actions within threads and non-overlapping method invocations. 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 (we provide a more detailed comparison below).

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. As we show in [7, §B, Decomposition Lemma], 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 $A = \{m = D_m \mid m \in M\}$. For a given n, the most general client $\mathsf{MGC}_n(A)$ is the combination of CFGs for the library A and those for the client with n threads that repeatedly invoke library methods in any order and with all possible parameters: the CFG for thread t is $(N_t, T_t, v^t_{\mathsf{mgc}}, v^t_{\mathsf{mgc}})$ with $N_t = \{v^t_{\mathsf{mgc}}\}$ and $T_t = \{(v^t_{\mathsf{mgc}}, m(u), v^t_{\mathsf{mgc}}) \mid m \in \mathsf{sig}(A), u \in \mathsf{Val}\}$. One can understand $\mathsf{MGC}_n(A)$ as "let A in $C^1_{\mathsf{mgc}} \parallel \ldots \parallel C^n_{\mathsf{mgc}}$" where C^t_{mgc} repeatedly makes all possible method calls. Let $N_{\mathsf{mgc}} = \biguplus_{t=1}^{n} N_t$.

We define the denotation of $\mathsf{MGC}_n(A)$ in a ***library-local semantics***, where the program executes only on the library part of state. Namely, we consider a relation $\longrightarrow_{\mathsf{MGC}_n(A)}$: $\mathsf{LConfig} \times \mathsf{Act} \times \mathsf{LConfig}$ transforming configurations $\mathsf{LConfig} = (\{1, \ldots, n\} \to \mathsf{LPos}) \times \mathsf{LState}$, where $\mathsf{LPos} = N_{\mathsf{mgc}} \cup (\mathsf{Val} \times N_{\mathsf{mgc}} \times \mathsf{LNode})$. The relation is defined as in Figure 1, but 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):

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

Let $[\![\mathsf{MGC}_n(A)]\!]_{\mathsf{lib}} \subseteq \mathsf{LTrace}$ be the set of all traces generated by the program $\mathsf{MGC}_n(A)$ 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 $[\![A]\!] = \bigcup_{n \geq 1} [\![\mathsf{MGC}_n(A)]\!]_{\mathsf{lib}}$. This lets us lift the notion of linearizability to libraries as follows.

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

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 starve actions. It assumes that the specification A_2 is sequential, meaning that every method in it is implemented by an atomic 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 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 only finite histories 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 to diverge and the execution of such methods to overlap with the executions of others. Our definitions require any method divergence 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 showing 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 $c_m; \mathsf{return}(E_m)$ for some $c_m \in \mathsf{PComm}$. Then the tools in [16, 15, 2] establish $A_1 \sqsubseteq A_2$, where $A_2 = \{m = (\mathsf{skip}^{\circlearrowleft}; c_m; \mathsf{skip}^{\circlearrowleft}; \mathsf{return}(E_m)) \mid m \in \mathsf{sig}(A_1)\}$. 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 $\mathsf{skip}^{\circlearrowleft}$ 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 corresponds to divergence at one of the $\mathsf{skip}^{\circlearrowleft}$ statements in our formalisation: discarding a pending call models the divergence at the first one, and completing a pending call with an arbitrary return 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. Namely, 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 $\tau, \tau' \in \mathsf{CTrace}$ are **equivalent**, written $\tau \sim \tau'$, if $\forall t \in \mathsf{ThreadID}.\ \tau|_t = \tau'|_t$ and there exists a bijection $\pi : \{1, \ldots, |\tau|\} \to \{1, \ldots, |\tau'|\}$ such*

that $\forall i.\, \tau(i) = \tau'(\pi(i))$ *and*

$$\forall i, j.\, (i < j \wedge \tau(i), \tau(j) \in \mathsf{ClientAct}) \Rightarrow \pi(i) < \pi(j)$$

Two traces are equivalent if they are permutations of each other preserving the order of actions within threads and all client actions. Note that $\mathsf{visible}(\tau) = \mathsf{visible}(\tau')$ 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*

$$\forall \tau_1 \in [\![C(A_1)]\!].\, \exists \tau_2 \in [\![C(A_2)]\!].\, \mathsf{client}(\tau_1) \sim \mathsf{client}(\tau_2) \wedge \mathsf{history}(\tau_1) \sqsubseteq \mathsf{history}(\tau_2).$$

Corollary 5. *If* $A_1 \sqsubseteq A_2$, *then* $\mathsf{visible}([\![C(A_1)]\!]) \subseteq \mathsf{visible}([\![C(A_2)]\!])$.

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*** (see, e.g., [12]), 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 $\{m = (\mathsf{skip}^{\circlearrowleft}; c_m; \mathsf{skip}^{\circlearrowleft}; \mathsf{return}(E_m)) \mid m \in \mathsf{sig}(A_1)\}$ for some atomic commands c_m and expressions E_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 starved 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* $P \subseteq \mathsf{History}$ *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 individual actions, as the latter can be rearranged during linearization. For example, the property "methods that are not starved by the scheduler always terminate" is linearization-closed. In contrast, the property "a $(t, \mathsf{ret}\ m(0))$ action is always followed by a $(t', \mathsf{ret}\ m(1))$ action" is not.

Corollary 7. *Let* $P \subseteq \mathsf{History}$ *be linearization-closed,* $A_1 \sqsubseteq A_2$, *and* $\mathsf{history}([\![A_1]\!]) \subseteq P$. *Then* $\mathsf{visible}([\![C(A_1)]\!]) \subseteq \mathsf{visible}([\![C(A_2)]\!] \cap \{\tau \mid \mathsf{history}(\tau) \in P\})$.

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 $[\![C(A_2)]\!] \cap \{\tau \mid \mathsf{history}(\tau) \in P\}$ 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, 4, 6] 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 starvation. The formal definition is as follows.

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

Here we use linear temporal logic (LTL) over histories, with predicates over actions as atomic propositions; $\Box$ 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 return infinitely often (for the case when the client calls infinitely many operations), or every operation that has not been starved by the scheduler return (for the case when the client calls only finitely many operations).

Note that the semantics of §2 allows for unfair schedulers that starve some threads. 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 starved. In Definition 8 we formalise it using starve actions.

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

```
struct Node {
  value_t data;
  Node *next;
};
Node *S;
int collision[SIZE];

void init() { S = NULL; }
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 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(); } }
```

Fig. 2. A non-blocking stack implementation

`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.

Lock-freedom of some algorithms, including Treiber's stack, can be proved automatically [6].

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 $\{m = (\mathsf{skip}^{\circlearrowleft}; c_m; \mathsf{skip}^{\circlearrowleft}; \mathsf{return}(E_m)) \mid m \in \mathsf{sig}(A)\}$, 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 $\mathsf{history}(\tau)$ 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 [6], which says that an algorithm is lock-free if and only if any number of its operations running in parallel do not have infinite traces, i.e., terminate if not starved. 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*

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

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

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 $\mathsf{skip}^{\circlearrowleft}$ 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 [7, §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 [6] 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. [5] have previously characterised linearizability in terms of observational refinement (technically, their result is similar to our Rearrangement Lemma in [7, §B]). They did not consider infinite computations and treated non-terminating methods approximately; thus, they could not handle liveness properties. Besides, 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, their formulation and 'proof' of this property are presented as a piece of informal text talking about artefacts without a clear semantics. As a consequence, their compositionality theorem misses an important requirement that library methods be linearizable.

Burckhardt et al. [3] 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 [12], which allows refining an atomic specification to a concurrent implementation. As atomicity refinement and abstractions are duals of each other, our results can also be used in the context of formal program development.

References

1. B. Alpern and F. B. Schneider. Defining liveness. *Inf. Process. Lett.*, 21(4), 1985.
2. D. Amit, N. Rinetzky, T. W. Reps, M. Sagiv, and E. Yahav. Comparison under abstraction for verifying linearizability. In *CAV*, 2007.
3. S. Burckhardt, C. Dern, M. Musuvathi, and R. Tan. Line-up: A complete and automatic linearizability checker. In *PLDI*, 2010.
4. B. Cook, A. Gotsman, A. Podelski, A. Rybalchenko, and M. Y. Vardi. Proving that programs eventually do something good. In *POPL*, 2007.
5. I. Filipović, P. O'Hearn, N. Rinetzky, and H. Yang. Abstraction for concurrent objects. In *ESOP*, 2009.
6. A. Gotsman, B. Cook, M. Parkinson, and V. Vafeiadis. Proving that non-blocking algorithms don't block. In *POPL*, 2009.

7. A. Gotsman and H. Yang. Liveness-preserving atomicity abstraction (extended version). Available from `www.software.imdea.org/~gotsman`, 2011.
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, 1990.
12. C. Jones. Splitting atoms safely. *TCS*, 375, 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*, 51, 1991.