A Generic Logic for Proving Linearizability

Artem Khyzha[1], Alexey Gotsman[1], and Matthew Parkinson[2]

[1] IMDEA Software Institute
[2] Microsoft Research Cambridge

Abstract. Linearizability is a commonly accepted notion of correctness for libraries of concurrent algorithms, and recent years have seen a number of proposals of program logics for proving it. Although these logics differ in technical details, they embody similar reasoning principles. To explicate these principles, we propose a logic for proving linearizability that is generic: it can be instantiated with different means of compositional reasoning about concurrency, such as separation logic or rely-guarantee. To this end, we generalise the Views framework for reasoning about concurrency to handle relations between programs, required for proving linearizability. We present sample instantiations of our generic logic and show that it is powerful enough to handle concurrent algorithms with challenging features, such as helping.

1 Introduction

To manage the complexity of constructing concurrent software, programmers package often-used functionality into *libraries* of concurrent algorithms. These encapsulate data structures, such as queues and lists, and provide clients with a set of methods that can be called concurrently to operate on these (e.g., `java.util.concurrent`). To maximise performance, concurrent libraries may use sophisticated non-blocking techniques, allowing multiple threads to operate on the data structure with minimum synchronisation. Despite this, each library method is usually expected to behave as though it executes atomically. This requirement is formalised by the standard notion of correctness for concurrent libraries, *linearizability* [14], which establishes a form of a simulation between the original *concrete* library and another *abstract* library, where each method is implemented atomically.

A common approach to proving linearizability is to find a *linearization point* for every method of the concrete library at which it can be thought of taking effect. [3] Given an execution of a concrete library, the matching execution of the abstract library, required to show the simulation, is constructed by executing the atomic abstract method at the linearization point of the concrete method. A difficulty in this approach is that linearization points are often not determined by a statically chosen point in the method code. For example, in concurrent algorithms with *helping* [13], a method may execute an operation originally requested by another method, called in a different thread; then the linearization point of the latter method is determined by an action of the former.

Recent years have seen a number of program logics for proving linearizability (see [6] for a survey). To avoid reasoning about the high number of possible interleavings between concurrently executing threads, these logics often use *thread-modular*

[3] Some algorithms cannot be reasoned about using linearization points, which we discuss in §7.

reasoning. They establish protocols that threads should follow when operating on the shared data structure and reason separately about every thread, assuming that the rest follow the protocols. The logics for proving linearizability, such as [26, 18], usually borrow thread-modular reasoning rules from logics originally designed for proving non-relational properties of concurrent programs, such as rely-guarantee [15], separation logic [21] or combinations thereof [26, 7]. Although this leads the logics to differ in technical details, they use similar methods for reasoning about linearizability, usually based on linearization points. Despite this similarity, designing a logic for proving linearizability that uses a particular thread-modular reasoning method currently requires finding the proof rules and proving their soundness afresh.

To consolidate this design space of linearization-point-based reasoning, we propose a logic for linearizability that is *generic*, i.e., can be instantiated with different means of thread-modular reasoning about concurrency, such as separation logic [21] or rely-guarantee [15]. To this end, we build on the recently-proposed Views framework [3], which unifies thread-modular logics for concurrency, such as the above-mentioned ones. Our contribution is to generalise the framework to reason about relations between programs, required for proving linearizability. In more detail, assertions in our logic are interpreted over a monoid of *relational views*, which describe relationships between the states of the concrete and the abstract libraries and the protocol that threads should follow in operating on these. The operation of the monoid, similar to the separating conjunction in separation logic [21], combines the assertions in different threads while ensuring that they agree on the protocols of access to the state. The choice of a particular relational view monoid thus determines the thread-modular reasoning method used by our logic.

To reason about linearization points, relational views additionally describe a set of special *tokens* (as in [26, 18, 2]), each denoting a one-time permission to execute a given atomic command on the state of the abstract library. The place where this permission is used in the proof of a concrete library method determines its linearization point, with the abstract command recorded by the token giving its specification. Crucially, reasoning about the tokens is subject to the protocols established by the underlying thread-modular reasoning method; in particular, their *ownership* can be transferred between different threads, which allows us to deal with helping.

We prove the soundness of our generic logic under certain conditions on its instantiations (Definition 2, §3). These conditions represent our key technical contribution, as they capture the essential requirements for soundly combining a given thread-modular method for reasoning about concurrency with the linearization-point method for reasoning about linearizability.

To illustrate the use of our logic, we present its example instantiations where thread-modular reasoning is done using disjoint concurrent separation logic [20] and a combination of separation logic and rely-guarantee [26]. We then apply the latter instantiation to prove the correctness of a sample concurrent algorithm with helping. We expect that our results will make it possible to systematically design logics using the plethora of other methods for thread-modular reasoning that have been shown to be expressible in the Views framework [20, 4, 1].

$$\longrightarrow \subseteq \mathsf{Com} \times \mathsf{PCom} \times \mathsf{Com}$$

$$\frac{C_1 \xrightarrow{\alpha} C_1'}{C_1 \ ; \ C_2 \xrightarrow{\alpha} C_1' \ ; \ C_2} \qquad \frac{}{C^\star \xrightarrow{\mathsf{id}} C; C^\star} \qquad \frac{}{\alpha \xrightarrow{\alpha} \mathsf{skip}}$$

$$\frac{i \in \{1,2\}}{C_1 + C_2 \xrightarrow{\mathsf{id}} C_i} \qquad \frac{}{\mathsf{skip} \ ; \ C \xrightarrow{\mathsf{id}} C} \qquad \frac{}{C^\star \xrightarrow{\mathsf{id}} \mathsf{skip}}$$

$$\longrightarrow\!\!\!\!\!\rightarrow \subseteq (\mathsf{Com} \times \mathsf{State}) \times (\mathsf{ThreadID} \times \mathsf{PCom}) \times (\mathsf{Com} \times \mathsf{State})$$

$$\frac{\sigma' \in [\![\alpha]\!]_t(\sigma) \quad C \xrightarrow{\alpha} C'}{\langle C, \sigma\rangle \xrightarrow{t,\alpha}\!\!\!\!\!\rightarrow \langle C', \sigma'\rangle}$$

Fig. 1. The operational semantics of sequential commands

2 Methods Syntax and Sequential Semantics

We consider concurrent programs that consist of two components, which we call *libraries* and *clients*. Libraries provide clients with a set of methods, and clients call them concurrently. We distinguish *concrete* and *abstract* libraries, as the latter serve as specification for the former due to its methods being executed atomically.

Syntax. Concrete methods are implemented as *sequential commands* having the syntax:

$$C \in \mathsf{Com} ::= \alpha \mid C \ ; \ C \mid C + C \mid C^\star \mid \mathsf{skip}, \quad \text{where } \alpha \in \mathsf{PCom}$$

The grammar includes *primitive commands* α from a set PCom, sequential composition $C \ ; \ C$, non-deterministic choice $C + C$ and a finite iteration $C^\star$ (we are interested only in terminating executions) and a termination marker skip. We use $+$ and $(\cdot)^\star$ instead of conditionals and while loops for theoretical simplicity: as we show at the end of this section, given appropriate primitive commands the conditionals and loops can be encoded. We also assume a set APCom of *abstract primitive commands*, ranged over by A, with which we represent methods of an abstract library.

Semantics. We assume a set State of concrete states of the memory, ranged over by σ, and abstract states AState, ranged over by Σ. The memory is shared among N threads with thread identifiers $\mathsf{ThreadID} = \{1, 2, \ldots, N\}$, ranged over by t.

We assume that semantics of each primitive command α is given by a non-deterministic *state transformer* $[\![\alpha]\!]_t : \mathsf{State} \to \mathcal{P}(\mathsf{State})$, where $t \in \mathsf{ThreadID}$. For a state σ, the set of states $[\![\alpha]\!]_t(\sigma)$ is the set of possible resulting states for α executed atomically in a state σ and a thread t. State transformers may have different semantics depending on a thread identifier, which we use to introduce thread-local memory cells later in the technical development. Analogously, we assume semantics of abstract primitive commands with state transformers $[\![\mathrm{A}]\!]_t : \mathsf{AState} \to \mathcal{P}(\mathsf{AState})$, all of which update abstract states atomically. We also assume a primitive command $\mathsf{id} \in \mathsf{PCom}$ with the interpretation $[\![\mathsf{id}]\!]_t(\sigma) \triangleq \{\sigma\}$, and its abstract counterpart $\mathsf{id} \in \mathsf{APCom}$.

The sets of primitive commands PCom and APCom as well as corresponding state transformers are parameters of our framework. In Figure 1 we give rules of operational semantics of sequential commands, which are parametrised by semantics of primitive commands. That is, we define a transition relation $\longrightarrow\!\!\!\!\!\rightarrow \subseteq (\mathsf{Com} \times \mathsf{State}) \times (\mathsf{ThreadID} \times \mathsf{PCom}) \times (\mathsf{Com} \times \mathsf{State})$, so that $\langle C, \sigma\rangle \xrightarrow{t,\alpha}\!\!\!\!\!\rightarrow \langle C', \sigma'\rangle$ indicates a tran-

sition from C to C' updating the state from σ to σ' with a primitive command α in a thread t. The rules of the operational semantics are standard.

Let us show how to define traditional control flow primitives, such as an if-statement and a while-loop, in our programming language. Assuming a language for arithmetic expressions, ranged over by E, and a function $[\![E]\!]_\sigma$ that evaluates expressions in a given state σ, we define a primitive command $\mathtt{assume}(E)$ that acts as a filter on states, choosing only those where E evaluates to non-zero values.

$$[\![\mathtt{assume}(E)]\!]_t(\sigma) \triangleq \text{ if } [\![E]\!]_\sigma \neq 0 \text{ then } \{\sigma\} \text{ else } \emptyset.$$

Using $\mathtt{assume}(E)$ and the C-style negation $!E$ in expressions, a conditional and a while-loop can be implemented as the following commands:

$$\mathtt{if}\ E\ \mathtt{then}\ C_1\ \mathtt{else}\ C_2 \triangleq (\mathtt{assume}(E); C_1) + (\mathtt{assume}(!E); C_2)$$
$$\mathtt{while}\ E\ \mathtt{do}\ C \triangleq (\mathtt{assume}(E); C)^\star; \mathtt{assume}(!E)$$

3 The generic logic

In this section, we present our framework for designing program logics for linearizability proofs. Given a concrete method and a corresponding abstract method, we aim to demonstrate that the former has a linearization point either within its code or in the code of another thread. The idea behind such proofs is to establish simulation between concrete and abstract methods using linearization points to determine when the abstract method has to make a transition to match a given execution of the concrete method. To facilitate such simulation-based proofs, we design our relational logic so that formulas in it denote relations between concrete states, abstract states and special *tokens*.

Tokens are our tool for reasoning about linearization points. At the beginning of its execution in a thread t, each concrete method m is given a token $\mathsf{todo}(\mathrm{A}_m)$ of the corresponding abstract primitive command A_m. The token represents a one-time permission for the method to take effect, i.e. to perform a primitive command A_m on an abstract machine. When the permission is used, a token $\mathsf{todo}(\mathrm{A}_m)$ in a thread t is irreversibly replaced with $\mathsf{done}(\mathrm{A}_m)$. Thus, by requiring that a method start its execution in a thread t with a token $\mathsf{todo}(\mathrm{A}_m)$ and ends with $\mathsf{done}(\mathrm{A}_m)$, we ensure that it has in its code a linearization point. The tokens of all threads are described by $\Delta \in \mathsf{Tokens}$:

$$\mathsf{Tokens} = \mathsf{ThreadID} \rightharpoonup (\{\mathsf{todo}(\mathrm{A}) \mid \mathrm{A} \in \mathsf{APCom}\} \cup \{\mathsf{done}(\mathrm{A}) \mid \mathrm{A} \in \mathsf{APCom}\})$$

Reasoning about states and tokens in the framework is done with the help of relational *views*. We assume a set Views, ranged over by p, q and r, as well as a reification function $\lfloor\ \rfloor : \mathsf{Views} \to \mathcal{P}(\mathsf{State} \times \mathsf{AState} \times \mathsf{Tokens})$ that interprets views as ternary relations on concrete states, abstract states and indexed sets of tokens.

Definition 1 *A relational view monoid is a commutative monoid* $(\mathsf{Views}, *, u)$*, where* Views *is an underlying set of relational views,* $*$ *is a monoid operation and* u *is a unit.*

The monoid structure of relational views allows treating them as restrictions on the environment of threads. Intuitively, each thread uses views to declare a protocol that other threads should follow while operating with concrete states, abstract states

and tokens. Similarly to the separating conjunction from separation logic, the monoid operation $*$ (view composition) applied to a pair of views combines protocols of access to the state and ensures that they do not contradict each other.

Disjoint Concurrent Separation logic. To give an example of a view monoid, we demonstrate the structure inspired by Disjoint Concurrent Separation logic (DCSL). A distinctive feature of DCSL is that its assertions enforce a protocol, according to which threads operate on disjoint pieces of memory. We assume a set of values Val, of which a subset $\mathsf{Loc} \subseteq \mathsf{Val}$ represents heap addresses. By letting $\mathsf{State} = \mathsf{AState} = (\mathsf{Loc} \rightharpoonup_{\mathrm{fin}} \mathsf{Val}) \cup \{\lightning\}$ we represent a state as either a finite partial function from locations to values or an exceptional faulting state $\lightning$, which denotes the result of an invalid memory access. We define an operation $\bullet$ on states, which results in $\lightning$ if either of the operands is $\lightning$, or the union of partial functions if their domains are disjoint. Finally, we assume that the set PCom consists of standard heap-manipulating commands with usual semantics [21, 3].

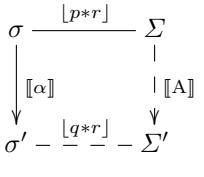
We consider the view monoid $(\mathcal{P}((\mathsf{State} \setminus \{\lightning\}) \times (\mathsf{AState} \setminus \{\lightning\}) \times \mathsf{Tokens}), *_{\mathsf{SL}}, ([\,], [\,], [\,]))$: the unit is a triple of nowhere defined functions $[\,]$, and the view composition defined as follows:

$$p *_{\mathsf{SL}} p' \triangleq \{(\sigma \bullet \sigma', \Sigma \bullet \Sigma', \Delta \uplus \Delta') \mid (\sigma, \Sigma, \Delta) \in p \wedge (\sigma', \Sigma', \Delta') \in p'\}.$$

In this monoid, the composition enforces a protocol of exclusive ownership of parts of the heap: a pair of views can be composed only if they do not simultaneously describe the content of the same heap cell or a token. Since tokens are exclusively owned in DCSL, they cannot be accessed by other threads, which makes it impossible to express a helping mechanism with the DCSL views. In §5, we present another instance of our framework and reason about helping in it.

Reasoning about linearization points. We now introduce *action judgements*, which formalise linearization-points-based approach to proving linearizability within our framework.

Let us assume that α is executed in a concrete state σ with an abstract state Σ and a set of tokens Δ satisfying a precondition p. According to the action judgement $\alpha \Vdash_t \{p\}\{q\}$, for every update $\sigma' \in [\![\alpha]\!]_t(\sigma)$ of the concrete state, the abstract state may be changed to $\Sigma' \in [\![\mathrm{A}]\!]_{t'}(\Sigma)$ in order to satisfy the postcondition q, provided that there is a token $\mathsf{todo}(\mathrm{A})$ in a thread t'. When the abstract state Σ is changed and the token $\mathsf{todo}(\mathrm{A})$ of a thread t' is used, the concrete state update corresponds to a linearization point, or to a regular transition otherwise.



Definition 2 *The action judgement $\alpha \Vdash_t \{p\}\{q\}$ holds, iff the following is true:*

$$\forall r, \sigma, \sigma', \Sigma, \Delta.\ (\sigma, \Sigma, \Delta) \in \lfloor p * r \rfloor \wedge \sigma' \in [\![\alpha]\!]_t(\sigma) \implies \exists \Sigma', \Delta'.\ \mathsf{LP}^*(\Sigma, \Delta, \Sigma', \Delta') \wedge (\sigma', \Sigma', \Delta') \in \lfloor q * r \rfloor,$$

where LP^* is the transitive closure of the following relation:

$$\mathsf{LP}(\Sigma, \Delta, \Sigma', \Delta') \triangleq \exists t', \mathrm{A}.\ \Sigma' \in [\![\mathrm{A}]\!]_{t'}(\Sigma) \wedge \Delta(t') = \mathsf{todo}(\mathrm{A}) \wedge \Delta' = \Delta[t' : \mathsf{done}(\mathrm{A})],$$

and $f[x : a]$ denotes the function such that $f[x : a](x) = a$ and for any $y \neq x$, $f[x : a](y) = f(y)$.

$$[\![\mathcal{P} * \mathcal{Q}]\!]_{\mathbf{i}} = [\![\mathcal{P}]\!]_{\mathbf{i}} * [\![\mathcal{Q}]\!]_{\mathbf{i}} \qquad [\![\mathcal{P} \Rrightarrow \mathcal{Q}]\!]_{\mathbf{i}} = [\![\mathcal{P}]\!]_{\mathbf{i}} \Rrightarrow [\![\mathcal{Q}]\!]_{\mathbf{i}}$$
$$[\![\mathcal{P} \vee \mathcal{Q}]\!]_{\mathbf{i}} = [\![\mathcal{P}]\!]_{\mathbf{i}} \vee [\![\mathcal{Q}]\!]_{\mathbf{i}} \qquad [\![\exists X.\, \mathcal{P}]\!]_{\mathbf{i}} = \bigvee_{n \in \mathsf{Val}} [\![\mathcal{P}]\!]_{\mathbf{i}[X:n]}$$

Fig. 2. Satisfaction relation for the assertion language Assn

Note that depending on pre- and postconditions p and q, $\alpha \Vdash_t \{p\}\{q\}$ may encode a regular transition, a conditional or a standard linearization point. It is easy to see that the latter is the case only when in all sets of tokens Δ from $\lfloor p \rfloor$ some thread t' has a todo-token, and in all Δ' from $\lfloor q \rfloor$ it has a done-token. Additionally, the action judgement may represent a conditional linearization point of another thread, as the LP relation allows using tokens of other threads.

Action judgements have a closure property that is important for thread-modular reasoning: when $\alpha \Vdash_t \{p\}\{q\}$ holds, so does $\alpha \Vdash_t \{p*r\}\{q*r\}$ for every view r. That is, execution of α and a corresponding linearization point preserves every view r that p can be composed with. Consequently, when in every thread action judgements hold of primitive commands and thread's views, all threads together mutually agree on each other's protocols of the access to the shared memory encoded in their views. This enables reasoning about every thread in isolation with the assumption that its environment follows its protocol. Thus, the action judgements formalise the requirements that instances of our framework need to satisfy in order to be sound. In this regard action judgements are inspired by semantic judgements of the Views Framework [3]. Our technical contribution is in formulating the essential requirements for thread-modular reasoning about linearizability of concurrent libraries with the linearization-point method and in extending the semantic judgement with them.

We let a *repartitioning implication* of views p and q, written $p \Rrightarrow q$, denote $\forall r.\, \lfloor p * r \rfloor \subseteq \lfloor q * r \rfloor$. A repartitioning implication $p \Rrightarrow q$ ensures that states satisfying p also satisfy q and additionally requires this property to preserve any view r.

Program logic. We are now in a position to present our generic logic for linearizability proofs via the linearization-point method. Assuming a view monoid and reification function as parameters, we define a minimal language Assn for assertions $\mathcal{P}$ and $\mathcal{Q}$ denoting sets of views:

$$\mathcal{P}, \mathcal{Q} \in \mathsf{Assn} ::= \rho \mid \mathcal{P} * \mathcal{Q} \mid \mathcal{P} \vee \mathcal{Q} \mid \mathcal{P} \Rrightarrow \mathcal{Q} \mid \exists X.\, \mathcal{P} \mid \ldots$$

The grammar includes view assertions ρ, a syntax VAssn of which is a parameter of the framework. Formulas of Assn may contain the standard connectives from separation logic, the repartitioning implication and the existential quantification over logical variables X, ranging over a set LVar.

Let us assume an interpretation of logical variables $\mathbf{i} \in \mathsf{Int} = \mathsf{LVar} \to \mathsf{Val}$ that maps logical variables from LVar to values from a finite set Val. In Figure 2, we define a function $[\![\cdot]\!]_{\cdot} : \mathsf{Assn} \times \mathsf{Int} \to \mathsf{Views}$ that we use to interpret assertions. Interpretation of assertions is parametrised by $[\![\cdot]\!]_{\cdot} : \mathsf{VAssn} \times \mathsf{Int} \to \mathsf{Views}$. In order to interpret disjunction, we introduce a corresponding operation on views and require the following properties from it:

$$\lfloor p \vee q \rfloor = \lfloor p \rfloor \cup \lfloor q \rfloor \qquad (p \vee q) * r = (p * r) \vee (q * r) \tag{1}$$

$$\text{(PRIM)}\ \frac{\forall \mathbf{i}.\ \alpha \Vdash_t \{[\![\mathcal{P}]\!]_{\mathbf{i}}\}\{[\![\mathcal{Q}]\!]_{\mathbf{i}}\}}{\vdash_t \{\mathcal{P}\}\ \alpha\ \{\mathcal{Q}\}} \qquad \text{(SEQ)}\ \frac{\vdash_t \{\mathcal{P}\}\ C_1\ \{\mathcal{P}'\} \quad \vdash_t \{\mathcal{P}'\}\ C_2\ \{\mathcal{Q}\}}{\vdash_t \{\mathcal{P}\}\ C_1\ ;\ C_2\ \{\mathcal{Q}\}}$$

$$\text{(FRAME)}\ \frac{\vdash_t \{\mathcal{P}\}\ C\ \{\mathcal{Q}\}}{\vdash_t \{\mathcal{P} * \mathcal{R}\}\ C\ \{\mathcal{Q} * \mathcal{R}\}} \qquad \text{(DISJ)}\ \frac{\vdash_t \{\mathcal{P}_1\}\ C\ \{\mathcal{Q}_1\} \quad \vdash_t \{\mathcal{P}_2\}\ C\ \{\mathcal{Q}_2\}}{\vdash_t \{\mathcal{P}_1 \vee \mathcal{P}_2\}\ C\ \{\mathcal{Q}_1 \vee \mathcal{Q}_2\}}$$

$$\text{(EX)}\ \frac{\vdash_t \{\mathcal{P}\}\ C\ \{\mathcal{Q}\}}{\vdash_t \{\exists X.\, \mathcal{P}\}\ C\ \{\exists X.\, \mathcal{Q}\}} \qquad \text{(CHOICE)}\ \frac{\vdash_t \{\mathcal{P}\}\ C_1\ \{\mathcal{Q}\} \quad \vdash_t \{\mathcal{P}\}\ C_2\ \{\mathcal{Q}\}}{\vdash_t \{\mathcal{P}\}\ C_1 + C_2\ \{\mathcal{Q}\}}$$

$$\text{(ITER)}\ \frac{\vdash_t \{\mathcal{P}\}\ C\ \{\mathcal{P}\}}{\vdash_t \{\mathcal{P}\}\ C^\star\ \{\mathcal{P}\}} \qquad \text{(CONSEQ)}\ \frac{\mathcal{P}' \Rrightarrow \mathcal{P} \quad \vdash_t \{\mathcal{P}\}\ C\ \{\mathcal{Q}\} \quad \mathcal{Q} \Rrightarrow \mathcal{Q}'}{\vdash_t \{\mathcal{P}'\}\ C\ \{\mathcal{Q}'\}}$$

Fig. 3. Proof rules

The judgements of the program logic take the form $\vdash_t \{\mathcal{P}\}\ C\ \{\mathcal{Q}\}$. In Figure 3, we present the proof rules, which are mostly standard. Among them, the PRIM rule is noteworthy, since it encorporates the simulation-based approach to reasoning about linearization points introduced by action judgements. The FRAME rule applies the idea of local reasoning from separation logic [21] to views. The CONSEQ enables weakening a precondition or a postcondition in a proof judgement and uses repartitioning implications to ensure the thread-modularity of the weakened proof judgement.

Semantics of proof judgements. We give semantics to judgements of the program logic by lifting the requirements of action judgements to sequential commands.

Definition 3 (Safety Judgement) *We define* safe_t *as the greatest relation such that the following holds whenever* $\mathsf{safe}_t(p, C, q)$ *does:*

- *if* $C \neq \mathsf{skip}$*, then* $\forall C', \alpha.\, C \xrightarrow{\alpha} C' \implies \exists p'.\ \alpha \Vdash_t \{p\}\{p'\} \wedge \mathsf{safe}_t(p', C', q)$*,*
- *if* $C = \mathsf{skip}$*, then* $p \Rrightarrow q$*.*

Lemma 4 $\forall t, \mathcal{P}, C, \mathcal{Q}.\ \vdash_t \{\mathcal{P}\}\ C\ \{\mathcal{Q}\} \implies \forall \mathbf{i}.\, \mathsf{safe}_t([\![\mathcal{P}]\!]_{\mathbf{i}}, C, [\![\mathcal{Q}]\!]_{\mathbf{i}})$.

We can understand the safety judgement $\mathsf{safe}_t([\![\mathcal{P}]\!]_{\mathbf{i}}, C, [\![\mathcal{Q}]\!]_{\mathbf{i}})$ as an obligation to create a sequence of views $[\![\mathcal{P}]\!]_{\mathbf{i}} = p_1, p_2, \ldots, p_{n+1} = [\![\mathcal{Q}]\!]_{\mathbf{i}}$ for each finite trace $\alpha_1, \alpha_2, \ldots, \alpha_n$ of C to justify each transition with action judgements $\alpha_1 \Vdash_t \{p_1\}\{p_2\}$, $\ldots$, $\alpha_n \Vdash_t \{p_n\}\{p_{n+1}\}$. Thus, when $\mathsf{safe}_t([\![\mathcal{P}]\!]_{\mathbf{i}}, C, [\![\mathcal{Q}]\!]_{\mathbf{i}})$ holds, it ensures that every step of the machine correctly preserves a correspondence between a concrete and abstract execution. Intuitively, the safety judgement lifts the simulation between concrete and abstract primitive commands established with action judgements to the implementation and specification of a method.

In Lemma 4, we establish that the proof judgements of the logic imply the safety judgements. As a part of the proof, we show that each of the proof rules of the logic holds of safety judgements. Due to space constraints, this and other proofs are given in the extended version of the paper [17].

4 Soundness

In this section, we formulate linearizability for libraries. We also formulate the soundness theorem, in which we state proof obligations that are necessary to conclude linearizability.

Libraries. We assume a set of method names Method, ranged over by m, and consider *a concrete library* $\ell : \mathsf{Method} \rightharpoonup ((\mathsf{Val} \times \mathsf{Val}) \to \mathsf{Com})$ that maps method names to commands from Com, which are parametrised by a pair of values from Val. For a given method name $m \in \mathsf{dom}(\ell)$ and values $a, v \in \mathsf{Val}$, a command $\ell(m, a, v)$ is an implementation of m, which accepts a as a method argument and either returns v or does not terminate. Such an unusual way of specifying method's arguments and return values significantly simplifies further development, since it does not require modelling a call stack.

Along with the library ℓ we consider its specification in the form of an abstract library $\mathcal{L} \in \mathsf{Method} \rightharpoonup ((\mathsf{Val}\times\mathsf{Val}) \to \mathsf{APCom})$ implementing a set of methods $\mathsf{dom}(\mathcal{L})$ atomically as abstract primitive commands $\{\mathcal{L}(m, a, v) \mid m \in \mathsf{dom}(\mathcal{L})\}$ parametrised by an argument a and a return value v. Given a method $m \in \mathsf{Method}$, we assume that a parametrised abstract primitive command $\mathcal{L}(m)$ is intended as a specification for $\ell(m)$.

Linearizability. The linearizability assumes a complete isolation between a library and its client, with interactions limited to passing values of a given data type as parameters or return values of library methods. Consequently, we are not interested in internal steps recorded in library computations, but only in the interactions of the library with its client. We record such interactions using *histories*, which are traces including only events $\mathsf{call}\ m(a)$ and $\mathsf{ret}\ m(v)$ that indicate an invocation of a method m with a parameter a and returning from m with a return value v, or formally:

$$h ::= \varepsilon \mid (t, \mathsf{call}\ m(a)) :: h \mid (t, \mathsf{ret}\ m(v)) :: h.$$

Given a library ℓ, we generate all finite histories of ℓ by considering N threads repeatedly invoking library methods in any order and with any possible arguments. The execution of methods is described by semantics of commands from § 2.

We define a *thread pool* $\tau : \mathsf{ThreadID} \to (\mathsf{idle} \uplus (\mathsf{Com} \times \mathsf{Val}))$ to characterise progress of methods execution in each thread. The case of $\tau(t) = \mathsf{idle}$ corresponds to no method running in a thread t. When $\tau(t) = (C, v)$, to finish some method returning v it remains to execute C.

Definition 5 *We let* $\mathcal{H}[\![\ell, \sigma]\!] = \bigcup_{n\geq 0} \mathcal{H}_n[\![\ell, (\lambda t.\, \mathsf{idle}), \sigma]\!]$ *denote the set of all possible histories of a library* ℓ *that start from a state* σ*, where for a given thread pool* τ*,* $\mathcal{H}_n[\![\ell, \tau, \sigma]\!]$ *is defined as a set of histories such that* $\mathcal{H}_0[\![\ell, \tau, \sigma]\!] \triangleq \{\varepsilon\}$ *and:*

$$\begin{aligned}
\mathcal{H}_n[\![\ell, \tau, \sigma]\!] \triangleq \{((t, \mathsf{call}\ m(a)) :: h) \mid\ & a \in \mathsf{Val} \wedge m \in \mathsf{dom}(\ell) \wedge \tau(t) = \mathsf{idle} \wedge \\
& \exists v.\, h \in \mathcal{H}_{n-1}[\![\ell, \tau[t : (\ell(m, a, v), v)], \sigma]\!]\} \\
\cup \{h \mid\ & \exists t, \alpha, C, C', \sigma', v.\, \tau(t) = (C, v) \wedge \langle C, \sigma\rangle \xrightarrow{t,\alpha}\!\!\!\!\rightarrow \langle C', \sigma'\rangle \wedge \\
& h \in \mathcal{H}_{n-1}[\![\ell, \tau[t : (C', v)], \sigma']\!]\} \\
\cup \{((t, \mathsf{ret}\ m(v)) :: h) \mid\ & m \in \mathsf{dom}(\ell) \wedge \tau(t) = (\mathsf{skip}, v) \wedge \\
& h \in \mathcal{H}_{n-1}[\![\ell, \tau[t : \mathsf{idle}], \sigma]\!]\}
\end{aligned}$$

Thus, we construct the set of all finite histories inductively with all threads initially idling. At each step of generation, in any idling thread t any method $m \in \mathsf{dom}(\ell)$ may be called with any argument a and an expected return value v, which leads to adding a command $\ell(m, a, v)$ to the thread pool of a thread t. Also, any thread t, in which $\tau(t) = (C, v)$, may do a transition $\langle C, \sigma\rangle \xrightarrow{t,\alpha}\!\!\!\!\rightarrow \langle C', \sigma'\rangle$ changing a command in the

thread pool and the concrete state. Finally, any thread that has finished execution of a method's command ($\tau(t) = (\mathsf{skip}, v)$) may become idle by letting $\tau(t) = \mathsf{idle}$.

We define $\mathcal{H}_n[\![\mathcal{L}, \mathcal{T}, \Sigma]\!]$ analogously and let the set of all histories of an abstract library $\mathcal{L}$ starting from the initial state Σ be $\mathcal{H}[\![\mathcal{L}, \Sigma]\!] = \bigcup_{n \geq 0} \mathcal{H}_n[\![\mathcal{L}, (\lambda t.\, \mathsf{idle}), \Sigma]\!]$.

Definition 6 *For libraries ℓ and $\mathcal{L}$ such that $\mathsf{dom}(\ell) = \mathsf{dom}(\mathcal{L})$, we say that $\mathcal{L}$ **linearizes** ℓ in the states σ and Σ, written $(\ell, \sigma) \sqsubseteq (\mathcal{L}, \Sigma)$, if $\mathcal{H}[\![\ell, \sigma]\!] \subseteq \mathcal{H}[\![\mathcal{L}, \Sigma]\!]$.*

That is, an abstract library $\mathcal{L}$ linearizes ℓ in the states σ and Σ, if every history of ℓ can be reproduced by $\mathcal{L}$. The definition is different from the standard one [14]: we use the result obtained by Gotsman and Yang [10] stating that the plain subset inclusion on the sets of histories produced by concrete and abstract libraries is equivalent to the original definition of linearizability.

Soundness w.r.t. linearizability. We now explain proof obligations that we need to show for every method m of a concrete library ℓ to conclude its linearizability. Particularly, for every thread t, argument a, return value v, and a command $\ell(m, a, v)$ we require that there exist assertions $\mathcal{P}(t, \mathcal{L}(m, a, v))$ and $\mathcal{Q}(t, \mathcal{L}(m, a, v))$, for which the following Hoare-style specification holds:

$$\vdash_t \{\mathcal{P}(t, \mathcal{L}(m, a, v))\}\ \ell(m, a, v)\ \{\mathcal{Q}(t, \mathcal{L}(m, a, v))\} \tag{2}$$

In the specification of $\ell(m, a, v)$, $\mathcal{P}(t, \mathcal{L}(m, a, v))$ and $\mathcal{Q}(t, \mathcal{L}(m, a, v))$ are assertions parametrised by a thread t and an abstract command $\mathcal{L}(m, a, v)$. We require that in a thread t of all states satisfying $\mathcal{P}(t, \mathcal{L}(m, a, v))$ and $\mathcal{Q}(t, \mathcal{L}(m, a, v))$ there be only tokens $\mathsf{todo}(\mathcal{L}(m, a, v))$ and $\mathsf{done}(\mathcal{L}(m, a, v))$ respectively:

$$\begin{array}{l} \forall \mathbf{i}, t, \sigma, \Sigma, \Delta, r. \\ \quad ((\sigma, \Sigma, \Delta) \in \lfloor [\![\mathcal{P}(t, \mathcal{L}(m, a, v))]\!]_{\mathbf{i}} * r \rfloor \implies \Delta(t) = \mathsf{todo}(\mathcal{L}(m, a, v))) \\ \quad \wedge\ ((\sigma, \Sigma, \Delta) \in \lfloor [\![\mathcal{Q}(t, \mathcal{L}(m, a, v))]\!]_{\mathbf{i}} * r \rfloor \implies \Delta(t) = \mathsf{done}(\mathcal{L}(m, a, v))) \end{array} \tag{3}$$

Together, (2) and (3) impose a requirement that a concrete and an abstract method return the same return value v. We also require that the states satisfying the assertions only differ by a token of a thread t:

$$\begin{array}{l} \forall \mathbf{i}, t, \mathrm{A}, \mathrm{A}', r, \Delta.\ (\sigma, \Sigma, \Delta[t : \mathsf{done}(\mathrm{A})]) \in \lfloor [\![\mathcal{Q}(t, \mathrm{A})]\!]_{\mathbf{i}} * r \rfloor \iff \\ \qquad\qquad (\sigma, \Sigma, \Delta[t : \mathsf{todo}(\mathrm{A}')]) \in \lfloor [\![\mathcal{P}(t, \mathrm{A}')]\!]_{\mathbf{i}} * r \rfloor. \end{array} \tag{4}$$

Theorem 7 *For given libraries ℓ and $\mathcal{L}$ together with states σ and Σ, $(\ell, \sigma) \sqsubseteq (\mathcal{L}, \Sigma)$ holds, if $\mathsf{dom}(\ell) = \mathsf{dom}(\mathcal{L})$ and (2), (3) and (4) hold for every method m, thread t and values a and v.*

5 The RGSep-based Logic

In this section, we demonstrate an instance of the generic proof system that is capable of handling algorithms with helping. This instance is based on RGSep [26], which combines rely-guarantee reasoning [15] with separation logic [21].

The main idea of the logic is to partition the state into several thread-local parts (which can only be accessed by corresponding threads) and the shared part (which

can be accessed by all threads). The partitioning is defined by proofs in the logic: an assertion in the code of a thread restricts its local state and the shared state. In addition, the partitioning is dynamic, meaning that resources, such as a part of a heap or a token, can be moved from the local state of a thread into the shared state and vice versa. By transferring a token to the shared state, a thread gives to its environment a permission to change the abstract state. This allows us to reason about environment helping that thread.

The RGSep-based view monoid. Similarly to DCSL, we assume that states represent heaps, i.e. that $\mathsf{State} = \mathsf{AState} = \mathsf{Loc} \rightharpoonup_{\mathrm{fin}} \mathsf{Val} \uplus \{\lightning\}$, and we denote all states but a faulting one with $\mathsf{State}_\mathsf{H} = \mathsf{AState}_\mathsf{H} = \mathsf{Loc} \rightharpoonup_{\mathrm{fin}} \mathsf{Val}$. We also assume a standard set of heap-manipulating primitive commands with usual semantics.

We define views as triples consisting of three components: a predicate P and binary relations R and G. A predicate $P \in \mathcal{P}((\mathsf{State}_\mathsf{H} \times \mathsf{AState}_\mathsf{H} \times \mathsf{Tokens})^2)$ is a set of pairs (l, s) of local and shared parts of the state, where each part consists of concrete state, abstract state and tokens. *Guarantee* G and *rely* R are relations from $\mathcal{P}((\mathsf{State} \times \mathsf{AState} \times \mathsf{Tokens})^2)$, which summarise how individual primitive commands executed by the method's thread (in case of G) and the environment (in case of R) may change the shared state. Together guarantee and rely establish a protocol that views of the method and its environment respectively must agree on each other's transitions, which allows us to reason about every thread separately without considering local state of other threads, assuming that they follow the protocol. The agreement is expressed with the help of a well-formedness condition on views of the RGSep-based monoid that their predicates must be *stable* under rely, meaning that their predicates take into account whatever changes their environment can make:

$$\mathsf{stable}(P, R) \triangleq \forall l, s, s'.\, (l, s) \in P \wedge (s, s') \in R \implies (l, s') \in P.$$

A predicate that is stable under rely cannot be invalidated by any state transition from rely. Stable predicates with rely and guarantee relations form the view monoid with the underlying set of views $\mathsf{Views}_{\mathsf{RGsep}} = \{(P, R, G) \mid \mathsf{stable}(P, R)\} \cup \{\bot\}$, where $\bot$ denotes a special *inconsistent* view with the empty reification. The reification of other views simply joins shared and local parts of the state:

$$\lfloor (P, R, G) \rfloor = \{(\sigma_l \bullet \sigma_s, \Sigma_l \bullet \Sigma_s, \Delta_l \uplus \Delta_s) \mid ((\sigma_l, \Sigma_l, \Delta_l), (\sigma_s, \Sigma_s, \Delta_s)) \in P\}.$$

Let an operation $\cdot$ be defined on states analogously to DCSL. Given predicates P and P', we let $P * P'$ be a predicate denoting the pairs of local and shared states in which the local state can be divided into two substates such that one of them together with the shared state satisfies P and the other together with the shared state satisfies P':

$$P{*}P' \triangleq \{((\sigma_l{\bullet}\sigma_l', \Sigma_l{\bullet}\Sigma_l', \Delta_l{\uplus}\Delta_l'), s) \mid ((\sigma_l, \Sigma_l, \Delta_l), s) \in P{\wedge}((\sigma_l', \Sigma_l', \Delta_l'), s) \in P'\}$$

We now define the monoid operation $*$, which we use to compose views of different threads. When composing views (P, R, G) and (P', R', G') of the parallel threads, we require predicates of both to be immune to interference by all other threads and each other. Otherwise, the result is inconsistent:

$$(P, R, G) * (P', R', G') \triangleq \text{if } G \subseteq R' \wedge G' \subseteq R \text{ then } (P * P', R \cap R', G \cup G') \text{ else } \bot.$$

$\models$: $(\mathsf{State} \times \mathsf{AState} \times \mathsf{Tokens}) \times (\mathsf{State} \times \mathsf{AState} \times \mathsf{Tokens}) \times \mathsf{Int} \times \mathsf{Assn}$

$((\sigma_l, \Sigma_l, \Delta_l), (\sigma_s, \Sigma_s, \Delta_s), \mathbf{i}) \models E \mapsto F$, iff $\sigma_l = [[\![E]\!]_{\mathbf{i}} : [\![F]\!]_{\mathbf{i}}]$, $\Sigma_l = [\,]$, and $\Delta_l = [\,]$

$((\sigma_l, \Sigma_l, \Delta_l), (\sigma_s, \Sigma_s, \Delta_s), \mathbf{i}) \models E \Mapsto F$, iff $\sigma_l = [\,]$, $\Sigma_l = [[\![E]\!]_{\mathbf{i}} : [\![F]\!]_{\mathbf{i}}]$, and $\Delta_l = [\,]$

$((\sigma_l, \Sigma_l, \Delta_l), (\sigma_s, \Sigma_s, \Delta_s), \mathbf{i}) \models [\mathsf{todo}(\mathrm{A})]_t$, iff $\sigma_l = [\,]$, $\Sigma_l = [\,]$, and $\Delta_l = [t : \mathsf{todo}(\mathrm{A})]$

$((\sigma_l, \Sigma_l, \Delta_l), (\sigma_s, \Sigma_s, \Delta_s), \mathbf{i}) \models [\mathsf{done}(\mathrm{A})]_t$, iff $\sigma_l = [\,]$, $\Sigma_l = [\,]$, and $\Delta_l = [t : \mathsf{done}(\mathrm{A})]$

$((\sigma_l, \Sigma_l, \Delta_l), (\sigma_s, \Sigma_s, \Delta_s), \mathbf{i}) \models \boxed{\pi}$, iff $\sigma_l = [\,]$, $\Sigma_l = [\,]$, $\Delta_l = [\,]$, and $((\sigma_s, \Sigma_s, \Delta_s), ([\,], [\,], [\,]), \mathbf{i}) \models \pi$

$((\sigma_l, \Sigma_l, \Delta_l), (\sigma_s, \Sigma_s, \Delta_s), \mathbf{i}) \models \pi * \pi'$, iff there exist $\sigma_l', \sigma_l'', \Sigma_l', \Sigma_l'', \Delta_l', \Delta_l''$ such that $\sigma_l = \sigma_l' \bullet \sigma_l''$, $\Sigma_l = \Sigma_l' \bullet \Sigma_l''$, $\Delta_l = \Delta_l' \uplus \Delta_l''$, $((\sigma_l', \Sigma_l', \Delta_l'), (\sigma_s, \Sigma_s, \Delta_s), \mathbf{i}) \models \pi$, and $((\sigma_l'', \Sigma_l'', \Delta_l''), (\sigma_s, \Sigma_s, \Delta_s), \mathbf{i}) \models \pi'$

Fig. 4. Satisfaction relation for a fragment of the assertion language VAssn

That is, we let the composition of views be consistently defined when the state transitions allowed in a guarantee of one thread are treated as environment transitions in the other thread, i.e. $G \subseteq R'$ and $G' \subseteq R$. The rely of the composition is $R \cap R'$, since the predicate $P * P'$ is guaranteed to be stable only under environment transitions described by both R and R'. The guarantee of the composition is $G \cup G'$, since other views need to take into account all state transitions either from G or from G'.

The RGSep-based program logic. We define the view assertion language VAssn that is a parameter of the proof system. Each view assertion ρ takes form of a triple $(\pi, \mathcal{R}, \mathcal{G})$, and the syntax for π is:

$$E ::= a \mid X \mid E + E \mid \ldots, \quad \text{where } X \in \mathsf{LVar}, a \in \mathsf{Val}$$
$$\pi ::= E = E \mid E \mapsto E \mid E \Mapsto E \mid [\mathsf{todo}(\mathrm{A})]_t \mid [\mathsf{done}(\mathrm{A})]_t \mid \boxed{\pi} \mid \pi * \pi \mid \neg\pi \mid \ldots$$

Formula π denotes a predicate of a view as defined by a satisfaction relation $\models$ in Figure 4. There $E \mapsto E$ and $E \Mapsto E$ denote a concrete and an abstract state describing singleton heaps. A non-boxed formula π denotes the view with the local state satisfying π and shared state unrestricted; $\boxed{\pi}$ denotes the view with the empty local state and the shared state satisfying π; $\pi * \pi'$ the composition of predicates corresponding to π and π'. The semantics of the rest of connectives is standard. Additionally, for simplicity of presentation of the syntax, we require that boxed assertions $\boxed{\pi}$ be not nested (as opposed to preventing that in the definition).

The other components $\mathcal{R}$ and $\mathcal{G}$ of a view assertion are sets of *rely/guarantee actions* $\mathcal{A}$ with the syntax: $\mathcal{A} ::= \pi \rightsquigarrow \pi'$. An action $\pi \rightsquigarrow \pi'$ denotes a change of a part of the shared state that satisfies π into one that satisfies π', while leaving the rest of the shared state unchanged. We associate with an action $\pi \rightsquigarrow \pi'$ all state transitions from the following set:

$$[\![\pi \rightsquigarrow \pi']\!] = \{((\sigma_s \bullet \sigma_s'', \Sigma_s \bullet \Sigma_s'', \Delta_s \uplus \Delta_s''), (\sigma_s' \bullet \sigma_s'', \Sigma_s' \bullet \Sigma_s'', \Delta_s' \uplus \Delta_s'')) \mid$$
$$\exists \mathbf{i}.\, (([\,], [\,], [\,]), (\sigma_s, \Sigma_s, \Delta_s), \mathbf{i}) \models \boxed{\pi} \wedge (([\,], [\,], [\,]), (\sigma_s', \Sigma_s', \Delta_s'), \mathbf{i}) \models \boxed{\pi'}\}$$

We give semantics to view assertions with the function $[\![\cdot]\!]_{\mathbf{i}}$ that is defined as follows:

$$[\![(\pi, \mathcal{R}, \mathcal{G})]\!]_{\mathbf{i}} \triangleq (\{(l, s) \mid (l, s, \mathbf{i}) \models \pi\}, \bigcup\nolimits_{\mathcal{A} \in \mathcal{R}} [\![\mathcal{A}]\!], \bigcup\nolimits_{\mathcal{A} \in \mathcal{G}} [\![\mathcal{A}]\!]).$$

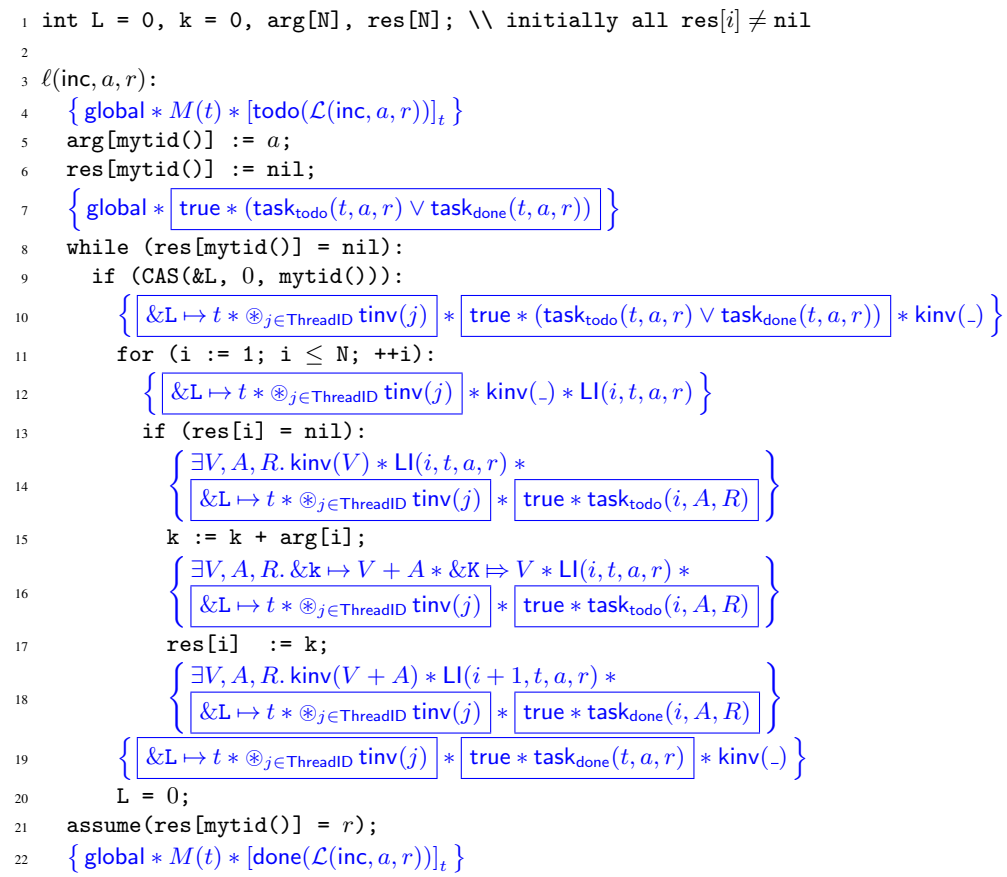


Fig. 5. Proof outline for a flat combiner of a concurrent increment. Indentation is used for grouping commands.

6 Example

In this section, we demonstrate how to reason about algorithms with helping using relational views. We choose a simple library ℓ implementing a concurrent increment and prove its linearizability with the RGSep-based logic.

The concrete library ℓ has one method inc, which increments the value of a shared counter k by the argument of the method. The specification of ℓ is given by an abstract library $\mathcal{L}$. The abstract command, provided by $\mathcal{L}$ as an implementation of inc, operates with an abstract counter K as follows (assuming that K is initialised by zero):

```
ℒ(inc, a, r):   < __kabs := __kabs + a; assume(__kabs == r); >
```

That is, $\mathcal{L}(\mathsf{inc}, a, r)$ atomically increments a counter and a command assume(K $==$ r), which terminates only if the return value r chosen at the invocation equals to the resulting value of K. This corresponds to how we specify methods' return values in §4.

In Figure 5, we show the pseudo-code of the implementation of a method inc in a C-style language along with a proof outline. The method $\ell(\mathsf{inc}, a, r)$ takes one argument,

$$
\begin{array}{ll}
X \not\mapsto Y & \triangleq \exists Y'.\, X \mapsto Y' * Y \neq Y' \\
M(t) & \triangleq \boxed{\mathsf{true} * (\&\mathtt{arg}[t] \mapsto \_ * \&\mathtt{res}[t] \not\mapsto \mathtt{nil})} \\
\mathsf{task}_{\mathsf{todo}}(t,a,r) & \triangleq \&\mathtt{arg}[t] \mapsto a * \&\mathtt{res}[t] \mapsto \mathtt{nil} * [\mathsf{todo}(\mathcal{L}(\mathsf{inc},a,r))]_t\,; \\
\mathsf{task}_{\mathsf{done}}(t,a,r) & \triangleq \&\mathtt{arg}[t] \mapsto a * \&\mathtt{res}[t] \mapsto r * r \neq \mathtt{nil} * [\mathsf{done}(\mathcal{L}(\mathsf{inc},a,r))]_t\,; \\
\mathsf{kinv}(V) & \triangleq \&\mathtt{k} \mapsto V * \&\mathtt{K} \Rrightarrow V \\
\mathsf{LI}(i,t,a,r) & \triangleq \boxed{\begin{array}{l}\mathsf{true} * ((t < i \wedge \mathsf{task}_{\mathsf{done}}(t,a,r)) \vee \\ (t \geq i \wedge (\mathsf{task}_{\mathsf{todo}}(t,a,r) \vee \mathsf{task}_{\mathsf{done}}(t,a,r))))\end{array}} \\
\mathsf{tinv}(i) & \triangleq \&\mathtt{arg}[i] \mapsto \_ * \&\mathtt{res}[i] \Rrightarrow \_ \vee \mathsf{task}_{\mathsf{todo}}(i,\_,\_) \vee \mathsf{task}_{\mathsf{done}}(i,\_,\_) \\
\mathsf{global} & \triangleq \boxed{(\&\mathtt{L} \mapsto 0 * \mathsf{kinv}(\_) \vee \&\mathtt{L} \not\mapsto 0) * \circledast_{j \in \mathsf{ThreadID}} \mathsf{tinv}(j)},
\end{array}
$$

Fig. 6. Auxiliary predicates. $\circledast_{j \in \mathsf{ThreadID}} \mathsf{tinv}(j)$ denotes $\mathsf{tinv}(1) * \mathsf{tinv}(2) * \cdot * \mathsf{tinv}(N)$

increments a shared counter `k` by it and returns the increased value of the counter. Since `k` is shared among threads, they follow a protocol regulating the access to the counter. This protocol is based on flat combining [11], which is a synchronisation technique enabling a parallel execution of sequential operations.

The protocol is the following. When a thread t executes $\ell(\mathsf{inc}, a, r)$, it first makes the argument of the method visible to other threads by storing it in an array `arg`, and lets $\mathtt{res}[t] = \mathtt{nil}$ to signal to other threads its intention to execute an increment with that argument. It then spins in the loop on line 8, trying to write its thread identifier into a variable `L` with a compare-and-swap (CAS). Out of all threads spinning in the loop, the one that succeeds in writing into `L` becomes a *combiner*: it performs the increments requested by all threads with arguments stored in `arg` and writes the results into corresponding cells of the array `res`. The other threads keep spinning and periodically checking the value of their cells in `res` until a non-`nil` value appears in it, meaning that a combiner has performed the operation requested and marked it as finished. The protocol relies on the assumption that `nil` is a value that is never returned by the method. Similarly to the specification of the increment method, the implementation in Figure 5 ends with a command $\mathtt{assume}(\mathtt{res}[\mathtt{mytid}()] = r)$.

The proof outline features auxiliary assertions defined in Figure 6. In the assertions we let $\_$ denote a value or a logical variable whose name is irrelevant. We assume that each program variable `var` has a unique location in the heap and denote it with $\&\mathtt{var}$. Values a, r and t are used in the formulas and the code as constants.

We prove the following specification for $\ell(\mathsf{inc}, a, r)$:

$$
\mathcal{R}_t, \mathcal{G}_t \vdash_t \left\{ \begin{array}{c} \mathsf{global} * M(t) * \\ [\mathsf{todo}(\mathcal{L}(\mathsf{inc},a,r))]_t \end{array} \right\} \ell(\mathsf{inc},a,r) \left\{ \begin{array}{c} \mathsf{global} * M(t) * \\ [\mathsf{done}(\mathcal{L}(\mathsf{inc},a,r))]_t \end{array} \right\}
$$

In the specification, $M(t)$ asserts the presence of $\mathtt{arg}[t]$ and $\mathtt{res}[t]$ in the shared state, and global is an assertion describing the shared state of all the threads. Thus, the pre- and postcondition of the specification differ only by the kind of token given to t.

The main idea of the proof is in allowing a thread t to share the ownership of its token $[\mathsf{todo}(\mathcal{L}(\mathsf{inc}, a, r))]_t$ with the other threads. This enables two possibilities for t. Firstly, t may become a combiner. Then t has a linearization point on line 17 (when the loop index i equals to t). In this case t also *helps* other concurrent threads by performing their linearization points on line 17 (when $i \neq t$). The alternative possibility is that some

other thread becomes a combiner and does a linearization point of t. Thus, the method has a non-fixed linearization point, as it may occur in the code of a different thread.

We further explain how the tokens are transferred. On line 6 the method performs the assignment `res[mytid()] := nil`, signalling to other threads about a task this thread is performing. At this step, the method transfers its token $[\mathsf{todo}(\mathcal{L}(\mathsf{inc}, a, r))]_t$ to the shared state, as represented by the assertion $\boxed{\mathsf{true} * \mathsf{task}_{\mathsf{todo}}(t, a, r)}$. In order to take into consideration other threads interfering with t and possibly helping it, here and further we stabilise the assertion by adding a disjunct $\mathsf{task}_{\mathsf{done}}(t, a, r)$.

If a thread t gets help from other threads, then $\mathsf{task}_{\mathsf{done}}(t, a, r)$ holds, which implies that $\mathtt{res}[t] \neq \mathtt{nil}$ and t cannot enter the loop on line 8. Otherwise, if t becomes a combiner, it transfers $\mathsf{kinv}(\_)$ from the shared state to the local state of t to take over the ownership of the counters `k` and `K` and thus ensure that the access to the counter is governed by the mutual exclusion protocol. At each iteration i of the forall loop, `res[i] = nil` implies that $\mathsf{task}_{\mathsf{todo}}(i, \_, \_)$ holds, meaning that there is a token of a thread i in the shared state. Consequently, on line 17 a thread t may use it to perform a linearization point of i.

The actions defining the guarantee relation $\mathcal{G}_t$ of a thread t' are the following:

1. $\&\mathtt{arg}[t] \mapsto \_ * \&\mathtt{res}[t] \not\mapsto \mathtt{nil} \rightsquigarrow \&\mathtt{arg}[t] \mapsto a * \&\mathtt{res}[t] \not\mapsto \mathtt{nil}$;
2. $\&\mathtt{arg}[t] \mapsto a * \&\mathtt{res}[t] \not\mapsto \mathtt{nil} \rightsquigarrow \mathsf{task}_{\mathsf{todo}}(t, a, r)$;
3. $\&\mathrm{L} \mapsto 0 * \mathsf{kinv}(\_) \rightsquigarrow \&\mathrm{L} \mapsto t$;
4. $\&\mathrm{L} \mapsto t * \mathsf{task}_{\mathsf{todo}}(T, A, R) \rightsquigarrow \&\mathrm{L} \mapsto t * \mathsf{task}_{\mathsf{done}}(T, A, R)$
5. $\&\mathrm{L} \mapsto t \rightsquigarrow \&\mathrm{L} \mapsto 0 * \mathsf{kinv}(\_)$
6. $\mathsf{task}_{\mathsf{done}}(t, a, r) \rightsquigarrow \&\mathtt{arg}[t] \mapsto a * \&\mathtt{res}[t] \mapsto r$

Out of them, conditions 2 and 6 specify transfering the token of a thread t to and from the shared state, and condition 4 describes using the shared token of a thread T. The rely relation of a thread t is then defined as the union of all actions from guarantee relations of other threads and an additional action for each thread $t' \in \mathsf{ThreadID} \setminus \{t\}$ allowing the client to prepare a thread t' for a new method call by giving it a new token: $[\mathsf{done}(\mathcal{L}(\mathsf{inc}, A, R))]_{t'} \rightsquigarrow [\mathsf{todo}(\mathcal{L}(\mathsf{inc}, A', R'))]_{t'}$.

7 Related Work

There has been a significant amount of research on methods for proving linearizability. Due to space constraints, we do not attempt a comprehensive survey here (see [6]) and only describe the most closely related work.

The existing logics for linearizability that use linearization points differ in the thread-modular reasoning method used and, hence, in the range of concurrent algorithms that they can handle. Our goal in this paper was to propose a uniform basis for designing such logics and to formalise the method they use for reasoning about linearizability in a way independent of the particular thread-modular reasoning method used. We have only shown instantiations of our logic based on disjoint concurrent separation logic [20] and RGSep [26]. However, we expect that our logic can also be instantiated with more complex thread-modular reasoning methods, such as those based on concurrent abstract predicates [4] or islands and protocols [25].

Our notion of tokens is based on the idea of treating method specifications as resources when proving atomicity, which has appeared in various guises in several logics [26, 18, 2]. Our contribution is to formalise this method of handling linearization points independently from the underlying thread-modular reasoning method and to formulate the conditions for soundly combining the two (Definition 2, §3).

We have presented a logic that unifies the various logics based on linearization points with helping. However, much work still remains as this reasoning method cannot handle all algorithms. Some logics have introduced *speculative* linearization points to increase their applicability [25, 18]; our approach to helping is closely related to this, and we hope could be extended to speculation. But there are still examples beyond this form of reasoning: for instance there are no proofs of the Herlihy-Wing queue [14] using linearization points (with helping and/or speculation). This algorithm can be shown linearizable using forwards/backwards simulation [14] and more recently has been shown to only require a backwards simulation [22]. But integrating this form of simulation with the more intrincate notions of interference expressible in the Views framework remains an open problem.

Another approach to proving linearizability is the aspect-oriented method. This gives a series of properties of a queue [12] (or a stack [5]) implementation which imply that the implementation is linearizable. This method been applied to algorithms that cannot be handled with standard linearization-point-based methods. However, the aspect-oriented approach requires a custom theorem per data structure, which limits its applicability.

In this paper we concentrated on linearizability in its original form [14], which considers only finite computations and, hence, specifies only safety properties of the library. Linearizability has since been generalised to also specify liveness properties [9]. Another direction of future work is to generalise our logic to handle liveness, possibly building on ideas from [19].

When a library is linearizable, one can use its atomic specification instead of the actual implementation to reason about its clients [8]. Some logics achieve the same effect without using linearizability, by expressing library specifications as judgements in the logic rather than as the code of an abstract library [16, 24, 23]. It is an interesting direction of future work to determine a precise relationship between this method of specification and linearizability, and to propose a generic logic unifying the two.

8 Conclusion

We have presented a logic for proving the linearizability of concurrent libraries that can be instantiated with different methods for thread-modular reasoning. To this end, we have extended the Views framework [3] to reason about relations between programs. Our main technical contribution in this regard was to propose the requirement for axiom soundness (Definition 2, §3) that ensures a correct interaction between the treatment of linearization points and the underlying thread-modular reasoning. We have shown that our logic is powerful enough to handle concurrent algorithms with challenging features, such as helping. More generally, our work marks the first step towards unifying the logics for proving relational properties of concurrent programs.

References

1. R. Bornat, C. Calcagno, P. W. O'Hearn, and M. J. Parkinson. Permission accounting in separation logic. In *POPL*, 2005.
2. P. da Rocha Pinto, T. Dinsdale-Young, and P. Gardner. TaDA: A logic for time and data abstraction. In *ECOOP*, 2014.
3. T. Dinsdale-Young, L. Birkedal, P. Gardner, M. J. Parkinson, and H. Yang. Views: compositional reasoning for concurrent programs. In *POPL*, 2013.
4. T. Dinsdale-Young, M. Dodds, P. Gardner, M. J. Parkinson, and V. Vafeiadis. Concurrent abstract predicates. In *ECOOP*, 2010.
5. M. Dodds, A. Haas, and C. M. Kirsch. A scalable, correct time-stamped stack. In *POPL*, New York, NY, USA, 2015.
6. B. Dongol and J. Derrick. Verifying linearizability: A comparative survey. *arXiv CoRR*, 1410.6268, 2014.
7. X. Feng. Local rely-guarantee reasoning. In *POPL*, 2009.
8. I. Filipovic, P. W. O'Hearn, N. Rinetzky, and H. Yang. Abstraction for concurrent objects. *Theoretical Computer Science*, 2010.
9. A. Gotsman and H. Yang. Liveness-preserving atomicity abstraction. In *ICALP*, 2011.
10. A. Gotsman and H. Yang. Linearizability with ownership transfer. *LMCS*, 2013.
11. D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *SPAA*, 2010.
12. T. A. Henzinger, A. Sezgin, and V. Vafeiadis. Aspect-oriented linearizability proofs. In *CONCUR*. 2013.
13. M. Herlihy and N. Shavit. *The art of multiprocessor programming*. 2008.
14. M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM TOPLAS*, 1990.
15. C. B. Jones. Specification and design of (parallel) programs. In *IFIP Congress*, 1983.
16. R. Jung, D. Swasey, F. Sieczkowski, K. Svendsen, A. Turon, L. Birkedal, and D. Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *POPL*, 2015.
17. A. Khyzha, A. Gotsman, and M. Parkinson. A generic logic for proving linearizability (extended version). *arXiv CoRR*, 1609.01171, 2016.
18. H. Liang and X. Feng. Modular verification of linearizability with non-fixed linearization points. In *PLDI*, 2013.
19. H. Liang, X. Feng, and Z. Shao. Compositional verification of termination-preserving refinement of concurrent programs. In *LICS*, 2014.
20. P. W. O'Hearn. Resources, concurrency, and local reasoning. *Theoretical Computer Science*, 2007.
21. P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *CSL*, 2001.
22. G. Schellhorn, H. Wehrheim, and J. Derrick. How to prove algorithms linearisable. In *CAV*. 2012.
23. I. Sergey, A. Nanevski, and A. Banerjee. Specifying and verifying concurrent algorithms with histories and subjectivity. In *ESOP*, 2015.
24. K. Svendsen and L. Birkedal. Impredicative concurrent abstract predicates. In *ESOP*, 2014.
25. A. J. Turon, J. Thamsborg, A. Ahmed, L. Birkedal, and D. Dreyer. Logical relations for fine-grained concurrency. In *POPL*, 2013.
26. V. Vafeiadis. Modular fine-grained concurrency verification. PhD Thesis. Technical Report UCAM-CL-TR-726, University of Cambridge, 2008.