# Theorems for Free from Separation Logic Specifications

LARS BIRKEDAL, Aarhus University, Denmark
THOMAS DINSDALE-YOUNG, Concordium, Denmark
ARMAËL GUÉNEAU, Aarhus University, Denmark
GUILHEM JABER, Université de Nantes, France
KASPER SVENDSEN, Uber, Denmark
NIKOS TZEVELEKOS, Queen Mary University of London, United Kingdom

Separation logic specifications with abstract predicates intuitively enforce a discipline that constrains when and how calls may be made between a client and a library. Thus a separation logic specification of a library intuitively enforces a protocol on the trace of interactions between a client and the library. We show how to formalize this intuition and demonstrate how to derive "free theorems" about such interaction traces from abstract separation logic specifications. We present several examples of free theorems. In particular, we prove that a so-called *logically atomic* concurrent separation logic specification of a concurrent module operation implies that the operation is linearizable. All the results presented in this paper have been mechanized and formally proved in the Coq proof assistant using the Iris higher-order concurrent separation logic framework.

**ACM Reference Format:**
Lars Birkedal, Thomas Dinsdale-Young, Armaël Guéneau, Guilhem Jaber, Kasper Svendsen, and Nikos Tzevelekos. 2021. Theorems for Free from Separation Logic Specifications. 1, 1 (June 2021), 29 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 INTRODUCTION

Separation logic [Ishtiaq and O'Hearn 2001; Reynolds 2002] provides a powerful formalism for specifying an interface between a library and a client in terms of resources. For example, a client may obtain an "opened file" resource by calling an `open` operation of a file library, which it can then use to access the file by calling a `read` operation. Crucially, abstract predicates [Biering et al. 2007; Parkinson and Bierman 2005] hide implementation details: a client cannot rely on what an "opened file" resource actually consists of; it can only rely on the functionality for using the resource that the library provides. A client that is verified with respect to the abstract specification will behave correctly with any implementation of the library.

In separation logic, functions are specified with preconditions and postconditions. We can think of the precondition as specifying resources that a client must provide in order to call the function. The postcondition then specifies the resources that the client receives when the function returns. For example, a simplified specification for a file library might look like:

$$\{\mathsf{closed}\}\ \mathtt{open}()\ \{\mathsf{open}\} \qquad \{\mathsf{open}\}\ \mathtt{close}()\ \{\mathsf{closed}\} \qquad \{\mathsf{open}\}\ \mathtt{read}()\ \{\mathsf{open}\}$$

Authors' addresses: Lars Birkedal, birkedal@cs.au.dk, Aarhus University, Denmark; Thomas Dinsdale-Young, tyoung@cs.au.dk, Concordium, Denmark; Armaël Guéneau, armael@cs.au.dk, Aarhus University, Denmark; Guilhem Jaber, guilhem.jaber@univ-nantes.fr, Université de Nantes, France; Kasper Svendsen, ksvendsen@cs.au.dk, Uber, Denmark; Nikos Tzevelekos, nikos.tzevelekos@qmul.ac.uk, Queen Mary University of London, United Kingdom.


XXXX-XXXX/2021/6-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

A client acquires an open resource, represented by an abstract predicate, by calling open. With this file resource, the client can call close and relinquish the resource, or call read and retain the resource. The rules of separation logic allow us to prove specifications for clients that use the library correctly, such as:

$$\{\texttt{closed}\}\ \texttt{open}();\ \texttt{read}();\ \texttt{close}()\ \{\texttt{closed}\}$$

On the other hand, we cannot prove any useful specifications[1] for programs that use the library incorrectly, like the following: open(); close(); read().

Intuitively, separation logic specifications imply properties about the trace of interactions between a library and a client. For example, the specification for the file library ensures that a file can only be accessed if it has previously been opened and not subsequently closed. This strongly depends on the fact that the specification is abstract: a client has no way to obtain the open resource except by calling the open operation. (Indeed, if the client were able to forge the open resource then it could violate the trace property.) In other words, an abstract separation logic specification of a library entails a theorem 'for free'; that is, a theorem that holds for *any* implementation of the library. While this intuition is broadly used in the separation logic community, it has not previously been formalised.

In this paper we present a formal approach to establishing free theorems from abstract separation logic specifications. In contrast to the well-known free theorems that one can obtain from polymorphic types [Reynolds and Plotkin 1993; Reynolds 1983; Wadler 1989], which are all concerned with equality properties (e.g., that every element of a polymorphic type is *equal to*[2] the polymorphic identity function), the free theorems we focus on here are all concerned with intensional trace properties describing how a library and a client interacts. Since the properties are intensional and not observable in the standard operational semantics, we establish the free theorems by placing a *wrapper* between a client and a library that generates a trace of the interactions between the client and library. We stress that the wrapper has no bearing on the underlying semantics of the program (when traces are ignored) but simply allows us to formally capture intensional trace properties. Then, supposing that a library implements an abstract separation logic specification, we show that the wrapped library also satisfies this specification and, moreover, maintains the desired trace property as an invariant. For the latter step, we use ghost trace resources, which can be used for instantiating the abstract predicates in the original specification. In the context of a client that uses the library in accordance with the specification, the trace properties are thus guaranteed to hold.

Our approach establishes *temporal* trace properties from separation logic specifications that are *not* inherently temporal, independently of implementation details. While ours is not the first approach incorporating temporal reasoning in program logics, and separation logic in particular, it is the first one to derive trace properties for libraries that have abstract separation logic specifications. Previous approaches [da Rocha Pinto et al. 2014; Fu et al. 2010; Gotsman et al. 2013; Sergey et al. 2015] achieve temporal reasoning by specifying and verifying the underlying libraries using trace-oriented specifications; the novelty of our work is in deriving temporal trace properties "for free" from trace-agnostic specifications.

We remark that informal English-language specifications (e.g., the POSIX file system [Gardner et al. 2014]) are often given in terms of trace properties, so the free theorems we obtain also serve to link formal separation logic specifications to more informal intuitive trace properties.

[1] It is always possible to prove a vacuous specification with precondition $\bot$, but a useful specification should at least have a satisfiable precondition. For complete, closed programs we would typically expect the precondition to be emp (denoting the empty heap), thus requiring the ownership of no specific resource.

[2] Be it in a denotational semantics model or contextual equivalence based on operational semantics.

We demonstrate our approach on a variety of examples, which establish that separation logic specifications can imply a variety of free theorems about interaction traces, such as:

- An iterator over a collection should only be used if the collection has not been modified since the iterator was created (§5.2).
- A higher-order function calls its argument exactly once (§5.3).
- A traversal on a stack invokes a given method on each value that has been pushed (but not popped) in the order in which they will be popped (§5.4).

As a more advanced example of a free theorem, we further prove that a *logically atomic* concurrent separation logic specification of a concurrent module operation [da Rocha Pinto et al. 2014; Jung et al. 2015a] implies that the operation is linearizable. To the best of our knowledge, this is the first formal such result. Note that this is indeed a free theorem in the sense it holds for *any* operation satisfying a logically atomic specification, i.e., the proof only relies on the specification and not on the specific operation.

We first outline our methodology on a motivating example in Section 2. We then formally define the programming language that we consider (Section 3), and the program logic that allows us to reason about interaction traces (Section 4). Then, we describe in Section 5 how our approach can be applied on a number of examples, culminating to the formal proof in Section 6 that logical atomicity implies linearizability, as a free theorem.

This work has been entirely mechanized in the Coq proof assistant using the Iris framework. It is publically available at either https://zenodo.org/record/4769405 (stable artifact) or https://github.com/logsem/free-theorems-sl (development repository).

## 2 MOTIVATING EXAMPLE

As a motivating example, consider a library that provides a stack with push and pop operations. In Separation Logic, these operations can be specified as follows:

$$\forall \alpha, x.\, \{\mathsf{stack}(\alpha) * \lceil x \neq ()\rceil\}\ \mathsf{push}(x)\ \{\mathsf{stack}(x :: \alpha)\}$$

$$\forall \alpha.\, \left\{\mathsf{stack}(\alpha)\right\}\ \mathsf{pop}()\ \left\{\begin{array}{c} r.\, (\lceil r = () \wedge \alpha = \varepsilon\rceil * \mathsf{stack}(\alpha)) \vee \\ (\exists \alpha'.\, \lceil \alpha = r :: \alpha'\rceil * \mathsf{stack}(\alpha')) \end{array}\right\}$$

The push operation simply prepends the given value to the stack, which is represented by the stack abstract predicate. The value must be distinct from the unit (), which is used to indicate an empty stack. The pop operation returns () if the stack is empty and otherwise removes and returns the head value. (We use $\lceil\cdot\rceil$ to inject pure propositions into the world of Separation Logic assertions.) The library would also include a way to create an initially empty stack, but we elide that here (see the more detailed treatment in §5).

We can also specify the stack in terms of trace properties that are satisfied by interactions between a client and a stack library. For instance, a simple trace property that we might wish to show is this:

> Each (non-unit) value returned by an invocation of pop was an argument of a previous invocation of push.

This property is not captured by most models of Separation Logic, in part simply because it is an intensional temporal trace property. We now outline how we can derive this property as a *free theorem*.

In order to capture trace properties, we define a wrapper that instruments the library operations with code to emit trace events annotated with push and pop labels:

$$\mathsf{push}' \triangleq \lambda v.\, \mathsf{push}(v); \mathsf{emit}\langle \mathsf{push}, v\rangle \qquad \mathsf{pop}' \triangleq \lambda \_.\, \mathsf{let}\ r = \mathsf{pop}()\ \mathsf{in}\ \mathsf{emit}\langle \mathsf{pop}, r\rangle; r$$

We can then define a language of traces that captures our desired invariant:

$$\mathcal{L} = \{t \mid \forall i, v.\, t[i] = \langle \mathsf{pop}, v\rangle \land v \neq () \implies \exists j.\, j < i \land t[j] = \langle \mathsf{push}, v\rangle\}$$

Our separation logic includes two resources that allow us to reason about code that emits trace events: $\mathsf{trace}(t)$ expresses that $t$ is the current trace; and $\mathsf{traceInv}(I)$ expresses that the trace invariant is $I$. The proof rule for emit updates the trace resource, while requiring that the new trace satisfies the invariant. Using these, we can define a 'wrapped' version of the abstract stack predicate so that the wrapped operations will satisfy the same abstract specification, but also enforce the trace invariant:

$$\mathsf{stack}'(\alpha) \triangleq \mathsf{stack}(\alpha) * \mathsf{traceInv}(\mathcal{L}) * \exists t.\, \mathsf{trace}(t) * \lceil t \in \mathcal{L} \land \forall x \in \alpha.\, \exists i.\, t[i] = \langle \mathsf{push}, x\rangle\rceil$$

Both the wrapped push and pop operations can be verified similarly. The proof of the wrapped push operation proceeds as follows:

---

$\{\mathsf{stack}'(\alpha) \land v \neq ()\}$
$\{\mathsf{stack}(\alpha) \land v \neq () * \mathsf{traceInv}(\mathcal{L}) * \exists t.\, \mathsf{trace}(t) * \lceil t \in \mathcal{L} \land \forall x \in \alpha.\, \exists i.\, t[i] = \langle \mathsf{push}, x\rangle\rceil\}$
  push(v);
$\{\mathsf{stack}(v :: \alpha) * \mathsf{traceInv}(\mathcal{L}) * \exists t.\, \mathsf{trace}(t) * \lceil t \in \mathcal{L} \land \forall x \in \alpha.\, \exists i.\, t[i] = \langle \mathsf{push}, x\rangle\rceil\}$
$\{\mathsf{stack}(v :: \alpha) * \mathsf{traceInv}(\mathcal{L}) * \exists t.\, \mathsf{trace}(t) * \lceil (t \cdot \langle \mathsf{push}, v\rangle) \in \mathcal{L}\rceil *$
    $\lceil \forall x \in (v :: \alpha).\, \exists i.\, (t \cdot \langle \mathsf{push}, v\rangle)[i] = \langle \mathsf{push}, x\rangle\rceil\}$
  **emit**⟨push,v⟩;
$\{\mathsf{stack}(v :: \alpha) * \mathsf{traceInv}(\mathcal{L}) * \exists t.\, \mathsf{trace}(t) * \lceil t \in \mathcal{L} \land \forall x \in \alpha.\, \exists i.\, t[i] = \langle \mathsf{push}, x\rangle\rceil\}$
$\{\mathsf{stack}'(v :: \alpha)\}$

---

Using an Adequacy theorem for our program logic, which links trace invariants to actual program executions (Theorem 4.1), we can thus conclude that the stack indeed satisfies the desired trace property. It is a free theorem because the proofs for the wrapped operations only depend on the specification for push and pop, not the actual code implementing the operations.

The above example demonstrates our technique on a first-order library (the library cannot make call-backs to the client), but it also applies in a higher-order setting. For instance, consider extending the stack with a `foreach` operation that traverses the stack and calls a client-supplied function on each element in order. In separation logic, this operation can be specified as:

$$\forall \alpha, f, I.\, \left\{\begin{array}{l}\mathsf{stack}(\alpha) * I(\varepsilon) * \forall \beta, x.\\ \{I(\beta)\}\ f(x)\ \{I(x :: \beta)\}\end{array}\right\}\ \mathtt{foreach}(f)\ \left\{\begin{array}{l}\mathsf{stack}(\alpha) *\\ I(\mathit{rev}(\alpha))\end{array}\right\}$$

This specification is subtle. The `foreach` operation takes a function $f$ that is specified (using a nested triple) with an invariant $I(\beta)$ whose parameter records the list of values that $f$ has so far been called on. The operation is given the predicate $I(\varepsilon)$ initially, and it returns with $I(\mathit{rev}(\alpha))$, where $\mathit{rev}(\alpha)$ is the reversal of the list $\alpha$. Note that while the predicate stack is abstract to the client (the library determines the interpretation), the predicate $I$ is abstract to the library (the client determines the interpretation). This means that for the library to obtain the $I(\mathit{rev}(\alpha))$ predicate for the postcondition from the $I(\varepsilon)$ given in the precondition, it must call $f$ on each element of $\alpha$ in order. Moreover, it cannot make any further calls to $f$, since separation logic treats the predicate $I(\beta)$ as a resource, which must be given up by the library each time it calls $f$, and which the library has no means of duplicating since in separation logic $A \implies A * A$ does not hold in general.

Defining a wrapper for higher-order libraries is more complex than in the first-order case, since we wish the trace to capture all interactions between the client and the library, including call-backs between them. The wrapper must therefore emit events at the call and return of library functions, as well as functions that are passed as arguments to library functions. We can then specify a number of properties that traces generated by client-library interactions will have:

- The trace of push and pop operations obeys the stack discipline.
- Between invocation and return, `foreach`($f$) calls $f$ on each element of the stack in order, with no further calls.
- The invocations of push, pop and $f$ (the argument of a call of `foreach`) are atomic — there are no further events between the call and return.

While the first property can be seen as a straightforward consequence of the push and pop specifications, the others are more subtle. In particular, they depend on the `foreach` specification's parametricity in $I$, which prevents the library from using its argument in an arbitrary fashion. For particular instantiations of $I$, (e.g. $I(\beta) = \mathsf{true}$), `foreach` could call its argument an arbitrary number of times, or even store it and call it from future invocations of pop. However, parametricity ensures that it will behave the same independent of how $I$ is instantiated, and so it cannot do that since $I$ can be instantiated so as to enforce trace properties.

This example illustrates several important aspects of our approach. Firstly, we support higher-order functions, such as `foreach`. We also deal with expressive trace properties: the language of traces is not context-free, since the stack may be traversed multiple times. Moreover, the connection between the separation logic specification and the trace properties that follow from it is subtle.

In §5 we revisit this example, among others, in detail. Before doing so, we present the formal setting of our approach.

## 3 PROGRAMMING LANGUAGE

We consider a fairly standard higher-order concurrent imperative language. We define the language in Figure 1: it is an untyped call-by-value $\lambda$-calculus with references, products and fork-based concurrency.

Additionally, our language features an emit primitive, which is used to output values as trace events, and a fresh primitive, which generates a fresh event tag and emits an event at the same time. These operations are used to implement the wrappers that instrument the operations of a given library so as to materialize the events of interest. We will formulate properties in terms of the event traces produced during program execution by these two primitives.

The definition of values (*Val*), expressions (*Exp*) and *evaluation contexts* (*Ectx*) is mostly standard. Most notable are the addition of the two primitives emit and fresh, and *tags* values $\tau$, which are simply represented as strings. Tags are used when defining wrappers so as to label emitted events and associate them with the corresponding function of a library interface. The fresh primitive can be used to emit an event annotated with a fresh tag, which is guaranteed not to already appear in the current trace. In simpler cases, the code of wrappers will simply refer to tag literals (i.e. strings) to decorate events emitted using emit. We will write `mytag` for a tag represented by the string "mytag". Importantly, tags can be passed around but cannot be compared, so they have no impact on the control flow of the program.

We equip the language with a small-step operational semantics. The head reduction judgment $e; \sigma \rightarrow_h e'; \sigma'; \overrightarrow{e_f}$ describes a single step of computation as can be performed by an individual thread. It expresses that, starting from state $\sigma$, the expression $e$ may reduce in one step to expression $e'$, while changing the state to $\sigma'$ and forking off a list of threads running expressions $\overrightarrow{e_f}$. A state $\sigma$ corresponds to a pair of a memory heap $h$ (a finite map from locations to values) and the current trace $t$ (a list of values). The threadpool reduction judgment $T; \sigma \longrightarrow T'; \sigma'$ lifts head reduction to lists of concurrently running threads.

Our language includes a compare-and-swap (CAS) primitive that can be used to implement locks or other concurrency primitives. Note that it is not terribly important what the exact set of features of our language is, as long as it is rich enough to allow for interesting implementations of the

$$\oplus ::= + \mid - \mid \times \mid = \mid <$$
$$v \in \mathit{Val} ::= \lambda x.e \mid \langle v, v\rangle \mid () \mid n \mid b \mid \ell \mid \tau$$
$$e \in \mathit{Exp} ::= v \mid x \mid \lambda x.e \mid e\ e \mid e \oplus e \mid \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \mid \langle e, e\rangle \mid \mathsf{fst}\ e \mid \mathsf{snd}\ e$$
$$\mid \mathsf{ref}\ e \mid\ !e \mid e \leftarrow e \mid \mathsf{fork}\ e \mid \mathsf{CAS}(e,e,e) \mid \mathsf{emit}\ e \mid \mathsf{fresh}\ e$$
$$K \in \mathit{Ectx} ::= (\text{standard left-to-right evaluation contexts})$$
$$n \in \mathbb{N},\ b \in \mathbb{B},\ \ell \in \mathit{Loc},\ \tau \in \mathit{Tag} \triangleq \mathit{String}$$

$$h \in \mathit{Heap} \triangleq \mathit{Loc} \rightharpoonup_{\mathrm{fin}} \mathit{Val} \qquad t \in \mathit{Trace} \triangleq \mathit{Val}^*$$
$$\sigma \in \mathit{State} \triangleq \mathit{Heap} \times \mathit{Trace} \qquad T \in \mathit{Tpool} \triangleq \mathit{Exp}^*$$
$$\mathrm{tags}(\varepsilon) \triangleq \varepsilon \qquad \mathrm{tags}(t \cdot \langle \tau, v\rangle) = \mathrm{tags}(t) \cdot \tau \qquad \mathrm{tags}(t \cdot v) = \mathrm{tags}(t) \ \text{ otherwise}$$

$\boxed{e;\sigma \to_h e';\sigma';\overrightarrow{e_f}}$ Head reduction (selected rules)

$$(\lambda x.e)\ v;\sigma \to_h e[v/x];\sigma;\varepsilon \qquad \mathsf{fork}\ e;\sigma \to_h ();\sigma;e$$
$$\mathsf{ref}\ v;(h,t) \to_h \ell;(h \uplus [l \mapsto v],t);\varepsilon \qquad \frac{\sigma.h(\ell) = v}{!\ell;\sigma \to_h v;\sigma;\varepsilon} \qquad \frac{h(\ell) = v}{\ell \leftarrow v';(h,t) \to_h ();(h[\ell \mapsto v'],t);\varepsilon}$$
$$\frac{\sigma.h(\ell) \neq v_1}{\mathsf{CAS}(\ell,v_1,v_2);\sigma \to_h \mathtt{false};\sigma;\varepsilon} \qquad \mathsf{CAS}(\ell,v_1,v_2);(h \uplus [\ell \mapsto v_1],t) \to_h \mathtt{true};(h \uplus [\ell \mapsto v_2],t);\varepsilon$$
$$\mathsf{emit}\ v;(h,t) \to_h ();(h,t \cdot v);\varepsilon \qquad \frac{\tau \notin \mathrm{tags}(t)}{\mathsf{fresh}\ v;(h,t) \to_h \tau;(h,t \cdot \langle \tau, v\rangle);\varepsilon}$$

$\boxed{T;\sigma \longrightarrow T';\sigma'}$ Threadpool reduction

$$\frac{e;\sigma \to_h e';\sigma';\overrightarrow{e_f}}{(T_1 +\!\!+ K[e] +\!\!+ T_2);\sigma \longrightarrow (T_1 +\!\!+ K[e'] +\!\!+ T_2 +\!\!+ \overrightarrow{e_f});\sigma'}$$

Fig. 1. Syntax and semantics of the language.

libraries and their clients for the specifications that we consider. In the rest of the paper, the code that we will directly verify is the code of wrappers, which only rely on fresh and emit, but it is important that we consider a language featureful enough for the whole setup to be realistic.

## 4 PROGRAM LOGIC

We use a program logic built as an instantiation of the Iris framework. For the purposes of this paper we present a particular instance of this framework for the language introduced in §3 and we refer to this instance simply as Iris.

### 4.1 Iris basics

Figure 2 describes the subset of Iris that is relevant for this paper. We write *iProp* for the universe of Iris propositions. These include the standard connectors of higher-order logic and separation logic, including the separationg conjunction $*$ and the magic wand $-\!\!*$ (also known as separating implication). The proposition $\lceil\phi\rceil$ asserts that the pure proposition $\phi$ holds, where $\phi$ is a proposition from the meta logic.

The modality $\Box$ is used to assert persistence of assertions. Iris assertions can be divided in two categories: *ephemeral* assertions and *persistent* assertions. Ephemeral assertions describe facts or

$$
\begin{array}{rll}
P, Q \in \mathit{iProp} ::= & \mathsf{True} \mid \mathsf{False} \mid \forall x.\, P \mid \exists x.\, P \mid \ldots & \text{higher-order logic} \\
& \mid P * Q \mid P \mathrel{-\!\!*} Q \mid \ulcorner \phi \urcorner \mid \Box P & \text{basic separation logic} \\
& \mid \{P\}\ e\ \{v.Q\} \mid \langle P \rangle\ e\ \langle v.Q \rangle & \text{program logic} \\
& \mid \boxed{P} \mid \ulcorner m \urcorner^{\gamma} & \text{invariants and ghost state} \\
& \mid \mathsf{trace}(t) \mid \mathsf{hist}(t) \mid \mathsf{traceInv}(\mathcal{L}) & \text{trace-related resources}
\end{array}
$$

Fig. 2. The subset of Iris relevant for this paper.

resources that are available at a given point but might become false or unavailable later. Persistent assertions describe facts that never cease to be true. The assertion $\Box P$ then holds if both $P$ holds and $P$ is persistent. Formally, we say that an assertion $P$ is persistent, written $\mathsf{persistent}(P)$, if $P \dashv\vdash \Box P$, where $\dashv\vdash$ is the logical equivalence of Iris propositions. As the knowledge associated with a persistent assertion can never be invalidated, persistent assertions can be freely duplicated.

Iris allows specifying programs: a Hoare triple $\{P\}\ e\ \{v.\, Q\}$ asserts that, when the precondition $P$ holds, then $e$ is safe to execute, and if $e$ reduces to a value $v$, then the postcondition $Q$ holds (note that $v$ acts as a binder in $Q$). Logically atomic triples $\langle P \rangle\ e\ \langle v.\, Q \rangle$ are used to specify concurrent operations, and are explained in more details in Section 6.1.

The assertion $\boxed{P}$ is an Iris invariant: it asserts that $P$ should hold at all times, and is therefore a persistent assertion. Iris also provides a way of tying the evolution of pure data $m$ to the execution of a program, through the use of ghost state, described by propositions $\ulcorner m \urcorner^{\gamma}$ (where $\gamma$ is a logical name identifying a piece of ghost state). We will only make use of invariants and (indirectly) ghost state in Section 6.

### 4.2 Trace primitives

Our instantiation of Iris is augmented with three (definable) assertions, $\mathsf{trace}(t)$, $\mathsf{hist}(t)$ and $\mathsf{traceInv}(\mathcal{L})$, for reasoning about the trace of events emitted during execution.

The resource $\mathsf{trace}(t)$ expresses that the trace of events emitted so far is exactly $t$ and asserts exclusive right to emit further trace events. Since $\mathsf{trace}(t)$ asserts exclusive right to emit events, it cannot be duplicated, and allows a single owner to reason precisely about the current trace.

The $\mathsf{trace}(t)$ resource suffices for examples where only a single resource needs to refer to the current trace. To improve expressiveness, we use the $\mathsf{hist}(t)$ resource to reason about prefixes of the current trace. The resource $\mathsf{hist}(t)$ asserts that the trace $t$ is a prefix of the trace of events emitted so far. Note that this property is preserved by emission of new events: if $t$ is a prefix of the current trace then $t$ is also a prefix of any extension of the current trace. This resource is persistent (and thus duplicable), and given the $\mathsf{trace}(t)$ resource we can construct a $\mathsf{hist}(t)$. Moreover, if we have a $\mathsf{trace}(t_1)$ resource and a $\mathsf{hist}(t_2)$ resource, we can conclude that $t_2$ is a prefix of $t_1$. These properties are given formally at the top of Figure 3.

Finally, the $\mathsf{traceInv}(\mathcal{L})$ resource defines a trace invariant that everyone must obey. Here $\mathcal{L}$ is a set of traces and $\mathsf{traceInv}(\mathcal{L})$ asserts that, at every point of the execution, the current trace must belong to $\mathcal{L}$. Since it specifies an invariant, the $\mathsf{traceInv}(\mathcal{L})$ resource is duplicable (see Figure 3). (Internally, traceInv is defined as an Iris invariant of the form $\boxed{\ldots}$, from which we prove as derived properties the simpler reasoning rules shown in Figure 3.)

We can now give Separation Logic specifications for the emit and fresh operations, which are shown in Figure 3. Both specifications require ownership over the trace resource $\mathsf{trace}(t)$, and that the trace is governed by some trace invariant $\mathcal{L}$. Given ownership of the trace resource we can

$$\mathsf{traceInv}(\mathcal{L}) \vdash \Box\,\mathsf{traceInv}(\mathcal{L}) \qquad \mathsf{hist}(t) \vdash \Box\,\mathsf{hist}(t) \qquad \mathsf{trace}(t) \vdash \mathsf{trace}(t) * \mathsf{hist}(t)$$

$$\mathsf{trace}(t_1) * \mathsf{hist}(t_2) \vdash \mathsf{trace}(t_1) * \lceil t_2 \leq_{pref} t_1 \rceil \qquad \mathsf{trace}(t) * \mathsf{traceInv}(\mathcal{L}) \vdash \mathsf{trace}(t) * \lceil t \in \mathcal{L} \rceil$$

$$\frac{t \cdot v \in \mathcal{L}}{\mathsf{traceInv}(\mathcal{L}) \vdash \{\mathsf{trace}(t)\}\ \mathsf{emit}\, v\ \{\mathsf{trace}(t \cdot v)\}}$$

$$\frac{\forall r.\, r \notin \mathsf{tags}(t) \Rightarrow t \cdot \langle r, v\rangle \in \mathcal{L}}{\mathsf{traceInv}(\mathcal{L}) \vdash \{\mathsf{trace}(t)\}\ \mathsf{fresh}\, v\ \{r.\, \lceil r \notin \mathsf{tags}(t)\rceil * \mathsf{trace}(t \cdot \langle r, v\rangle)\}}$$

Fig. 3. Properties of the resources trace, hist and traceInv, and specifications for emit and fresh.

emit an event $v$ which is appended at the end of the trace. Emitting a fresh event $v$ returns a tag $r$ that does not appear in the current trace, and emits the event $\langle r, v\rangle$.

As a side-condition, the updated trace is required to satisfy the invariant $\mathcal{L}$ (in the case of fresh, it is required to satisfy $\mathcal{L}$ for any fresh tag, since we cannot know in advance the tag that will be returned by fresh). Finally, both specifications hand out an updated trace resource in the postcondition, with the emitted event appended at the end.

We emphasize that the predicates and the properties in Figure 3 are all defined and proved formally in Iris, see the accompanying Coq formalization for details.

### 4.3 Adequacy theorem

Ultimately, a program logic is only a technical device: the end goal is to establish properties that can be shown to hold with respect to the operational semantics of the language, independently of the details of the logic. This is typically achieved by proving an "adequacy" or "soundness" theorem of the logic. We present one such theorem, Theorem 4.1, tailored to our goal of establishing trace properties for the interaction between a verified library and its client.

Theorem 4.1 (Adequacy). *Given any specification $\Phi$, resource $P_0$, initialiser $e_{init}$, library implementation $v_{lib}$, memory heap $h$, client $e$, and language $\mathcal{L}$, if the following conditions hold:*

- $\Phi : iProp \times Val \to iProp$ *and* $P_0 : iProp$
- $\varepsilon \in \mathcal{L}$
- $\forall lib, P.\ \Phi(P, lib) \vdash \{P\}\ e\ lib\ \{\top\}$
- $\vdash \{\top\}\ e_{init}\ \{P_0\}$
- $\vdash \Phi(P_0 * \mathsf{trace}(\varepsilon) * \mathsf{traceInv}(\mathcal{L}), v_{lib})$

*then for any $T, h', t$, provided $(e_{init}; e\, v_{lib}), (h, \varepsilon) \longrightarrow^* T, (h', t)$ then we have $t \in \mathcal{L}$.*

To understand the theorem, first recall that the general pattern we follow is to prove that for any library implementation satisfying an abstract library specification, the wrapped library implementation satisfies the same abstract specification and moreover the traces generated by the wrapping satisfy a given invariant. Client programs verified against the abstract library interface can thus be linked with the wrapped library implementation to conclude that the traces generated by the wrapping satisfy the given invariant. Theorem 4.1 formalises this idea. Here $e_{init}$ is an initialiser operation to initialize the internal state $P_0$ of the library. The theorem requires that we provide a library implementation satisfying the abstract specification $\Phi$ and generating traces in $\mathcal{L}$. It then guarantees that, for any client $e$ verified against the abstract specification of the library, the

trace produced by running the client linked with the library implementation $v_{lib}$ satisfies the trace invariant $\mathcal{L}$ *at every step of the execution.*

It is worth noting that the assumptions of Theorem 4.1 rules out clients containing calls to emit or fresh, which could in principle invalidate the trace property. Indeed, following assumption 3 of the theorem, we only consider *verified* clients $e$ that can be proved safe assuming an arbitrary proposition $P$ as pre-condition and the specification of the library $\Phi(P, lib)$. Neither of these propositions can give access to the $\mathsf{trace}(\cdot)$ and $\mathsf{traceInv}(\cdot)$ predicates necessary to verify calls to emit and fresh (we have to establish $\Phi$ in assumption 5 which prevents it from providing these predicates for any $P$).

To use Theorem 4.1 to derive a trace property from a separation logic specification, the idea is to define a suitable wrapper function **wrap** for the library in question and prove that if a library implementation $M$ satisfies $\Phi$ then so does the wrapped version $\mathbf{wrap}(M)$ *and*, additionally, the wrapped version generates traces in the $\mathcal{L}$ language:

$$\forall P_0, M.\ \ \Phi(P_0, M) \vdash \Phi(P_0 * \mathsf{trace}(\varepsilon) * \mathsf{traceInv}(\mathcal{L}), \mathbf{wrap}(M)). \tag{1}$$

One can then apply Theorem 4.1 by taking $v_{lib}$ to be $\mathbf{wrap}(M)$. For each of the examples coming up next, we will establish a lemma of the form (1). We stress that each of these results is not just a fact that holds in our logic: each one can be composed with Theorem 4.1 to obtain that the trace property of interest is preserved *at the level of the operational semantics*. (We refer to the Coq formalization for the details of each specific instantiation.) We also emphasize that (1) is shown for any $M$ and thus the trace property only depends on the specification $\Phi$ and in that sense we obtain a free theorem.

## 5 PROVING TRACE PROPERTIES AS FREE THEOREMS

In this section, we demonstrate the approach sketched earlier through a series of increasingly complex examples, starting with the basic file library example from the introduction.

It is worth noting that, even though we work in a concurrent language, the examples presented in this section are sequential in nature: the specifications considered prevent concurrent calls to the underlying library. We will look more closely into the case of concurrent libraries in Section 6.

### 5.1 File Library

Recall the Separation Logic specification of the file library from the introduction, here written more formally, where $P_0$ denotes the resources needed to initialize the library.

$$\Phi_{\text{file}}(P_0, (\mathit{open}, \mathit{close}, \mathit{read})) \triangleq \exists \mathsf{open}, \mathsf{closed} : \mathit{iProp}.\ \square(P_0 \mathrel{-\!\!*} \mathsf{closed}) \wedge \\ \{\mathsf{closed}\}\ \mathit{open}()\ \{\mathsf{open}\} \wedge \{\mathsf{open}\}\ \mathit{close}()\ \{\mathsf{closed}\} \wedge \{\mathsf{open}\}\ \mathit{read}()\ \{\mathsf{open}\}$$

Note the use of different fonts: *open*, *close* and *read* denote functions (values of the language), while open and closed denote abstract Separation Logic predicates used in the specification of the functions. Below we will also use the `open`, `close` and `read` tags (recall that they correspond to mere strings) for writing the wrapper code.

We wish to prove that this Separation Logic specification enforces that verified clients can only close and read when the file is open. To capture this property, we define a wrapper function $\mathbf{wrap}_{\text{file}}$ that instruments an implementation of the file library to emit events annotating calls to the different operations with respective tags `open`, `close` and `read`. Formally, we take a file library to be a triple consisting of an *open*, a *close* and a *read* function.

$$\mathbf{wrap}_{\text{file}} \triangleq \lambda(\mathit{open}, \mathit{close}, \mathit{read}).\\ \left(\lambda\_.\ \mathit{open}(); \mathsf{emit}\ \mathtt{open},\ \lambda\_.\ \mathit{close}(); \mathsf{emit}\ \mathtt{close},\ \lambda\_.\ \mathit{read}(); \mathsf{emit}\ \mathtt{read}\right)$$

We can now formalize the protocol as constraints on the traces generated by linking an instrumented file library with a client. In particular, we consider traces belonging to the language $\mathcal{L}_{\text{file}}$ of all strings $t \in \Sigma^*$ such that $t$ is a valid file trace, $\text{file}_{\text{trace}}(t)$, where $\Sigma = \{\text{open}, \texttt{close}, \texttt{read}\}$ and $\text{file}_{\text{trace}}$ is defined as:

$$\text{file}_{\text{trace}}(t) \triangleq \forall n.\, t[n] = \texttt{read} \vee t[n] = \texttt{close} \implies \text{isopen}(t, n)$$
$$\text{isopen}(t, n) \triangleq \exists m < n.\, t[m] = \text{open} \wedge \text{noclose}(t, m, n)$$
$$\text{noclose}(t, m, n) \triangleq \forall k.\, m < k < n \implies t[k] \neq \texttt{close}$$

The valid file trace predicate, $\text{file}_{\text{trace}}(t)$, expresses that the trace $t$ only contains read and close events when the file is open. To prove that the specification enforces the intended protocol, we proceed by proving that the wrapping preserves satisfaction of the specification and generates traces in $\mathcal{L}_{\text{file}}$.

Lemma 5.1. $\forall P_0, \mathit{ops}.\ \Phi_{\mathit{file}}(P_0, \mathit{ops}) \vdash \Phi_{\mathit{file}}(P_0 * \text{trace}(\varepsilon) * \text{traceInv}(\mathcal{L}_{\text{file}}), \mathbf{wrap}_{\mathit{file}}(\mathit{ops}))$.

Proof Sketch. We first need to define new wrapped versions of the abstract representation predicates, which relate the abstract resources to the current trace. The idea is that the wrapped open resource, $\text{open}_{\text{w}}$, should express that the current trace $t$ is in $\mathcal{L}_{\text{file}}$ and that the trace $t$ is open. Likewise, the wrapped closed resource, $\text{closed}_{\text{w}}$ can simply express that the current trace is in $\mathcal{L}_{\text{file}}$.

$$\text{open}_{\text{w}} \triangleq \text{open} * \exists t.\, \text{trace}(t) * \text{traceInv}(\mathcal{L}_{\text{file}}) * \lceil \text{isopen}(t, |t|) \rceil$$
$$\text{closed}_{\text{w}} \triangleq \text{closed} * \exists t.\, \text{trace}(t) * \text{traceInv}(\mathcal{L}_{\text{file}})$$

We need to prove $P_0 * \text{trace}(\varepsilon) * \text{traceInv}(\mathcal{L}_{\text{file}}) \mathrel{-\!\!*} \text{closed}_{\text{w}}$ assuming $P_0 \mathrel{-\!\!*} \text{closed}$, which follows trivially from the definition of $\text{closed}_{\text{w}}$. It remains to be proved that the wrapped methods satisfy their specifications.

We give on the right a proof outline for the wrapped open method. Here we use the assumed specification of the underlying open method to verify the call to open, and we used the following property of $\mathcal{L}_{\text{file}}$ and isopen to prove that emitting open would result in a trace in $\mathcal{L}_{\text{file}}$ that was open (recall that trace($t$) combined with traceInv($\mathcal{L}_{\text{file}}$) entails $t \in \mathcal{L}_{\text{file}}$):

$\{\text{closed}_{\text{w}}\}$
$\{\text{closed} * \exists t.\, \text{trace}(t) * \text{traceInv}(\mathcal{L}_{\text{file}})\}$
  open();
$\{\text{open} * \exists t.\, \text{trace}(t) * \text{traceInv}(\mathcal{L}_{\text{file}})\}$
  **emit** open;
$\{\text{open} * \exists t.\, \text{trace}(t) * \text{traceInv}(\mathcal{L}_{\text{file}}) *$
  $\lceil \text{isopen}(t, |t|) \rceil\}$
$\{\text{open}_{\text{w}}\}$

$$\forall t \in \mathcal{L}_{\text{file}}.\ t \cdot \text{open} \in \mathcal{L}_{\text{file}} \wedge \text{isopen}(t \cdot \text{open}, |t \cdot \text{open}|).$$

The proofs for the close and read operations are similar, but rely on the following property to justify the emits:

$$\forall t \in \mathcal{L}_{\text{file}}.\, \text{isopen}(t, |t|) \implies t \cdot \texttt{close} \in \mathcal{L}_{\text{file}} \wedge t \cdot \texttt{read} \in \mathcal{L}_{\text{file}}.$$

By combining Lemma 5.1 with the Adequacy theorem (Theorem 4.1) we have thus shown the free theorem that any verified client can only close and read when the file is open.

### 5.2 Iterators on Collections

We consider a collections library that provides methods for modifying a collection as well as iterating over it. The combination of these two features means that we have to be wary of *iterator invalidation*: it is typically unsafe to use an iterator after the collection it is iterating over has been modified. Following Krishnaswami et al. [2009], we assume for such a library a cut-down Separation Logic library specification that does not track the contents of the underlying collections, and consists of five operations. The *size* operation is non-destructive and returns the size of the

given collection; the *add* and *remove* operations destructively modify the given collection by adding or removing an element; and finally, *iterator* returns a new iterator for the collection, while *next* returns the next element of a given iterator.

$$\begin{aligned}
&\Phi_{\text{coll}}(P_0, (\mathit{size}, \mathit{add}, \mathit{remove}, \mathit{iterator}, \mathit{next})) \triangleq \\
&\quad \exists \text{coll} : \mathit{Val} \to \mathit{iProp}.\ \exists \text{iter} : \mathit{Val} \times \mathit{Val} \to \mathit{iProp}.\ \Box(P_0 \mathrel{-\!\!*} \exists c : \mathit{Val}.\ \text{coll}(c)) \land \\
&\qquad \forall c : \mathit{Val}.\ \{\text{coll}(c)\}\ \mathit{size}()\ \{x.\ \text{coll}(c)\}\ \land \\
&\qquad \forall c, x : \mathit{Val}.\ \{\text{coll}(c)\}\ \mathit{add}(x)\ \{\exists c' : \mathit{Val}.\ \text{coll}(c')\}\ \land \\
&\qquad \forall c, x : \mathit{Val}.\ \{\text{coll}(c)\}\ \mathit{remove}(x)\ \{\exists c' : \mathit{Val}.\ \text{coll}(c')\}\ \land \\
&\qquad \forall c : \mathit{Val}.\ \{\text{coll}(c)\}\ \mathit{iterator}()\ \{r.\ \text{coll}(c) * \text{iter}(r, c)\}\ \land \\
&\qquad \forall c, x : \mathit{Val}.\ \{\text{coll}(c) * \text{iter}(x, c)\}\ \mathit{next}(x)\ \{\text{coll}(c) * \text{iter}(x, c)\}
\end{aligned}$$

A seasoned practicioner of Separation Logic will be able to read from this specification that it successfully prevents the pitfall of iterator invalidation, i.e. it prevents the use of iterators that have been invalidated by a modification to the collection. This is not completely obvious, and comes from the combination of three observations. Firstly, the $\text{coll}(c)$ resource, which represents the ownership over the collection, is indexed by a value $c$ which can be seen as an abstract version number. Secondly, destructive operations such as *add* or *remove* update the version number to a new *existentially quantified* version number, which can only be assumed to be distinct from all the previous versions of the collection. Finally, the $\text{iter}(p, c)$ resource expresses that $p$ is an iterator over a collection which was at version $c$ when the iterator was created; the specification of *next* then requires the ownership of both an iterator and its collection with matching version numbers.

In other words, by simply reading the specification we can deduce the following free theorem:

> *An iterator over a collection can only be used if the underlying collection has not been destructively modified since the iterator was created.*

This is a trace property of the interaction between the collection library and clients. To capture it formally, we define as before a wrapper for the library that produces appropriate trace events. The instrumentation is fairly straightforward and simply emits a suitable event indicating the operation called and the argument and/or return value of the given operation, when relevant:

$$\mathbf{wrap}_{\text{coll}} \triangleq \lambda(\mathit{size}, \mathit{add}, \mathit{remove}, \mathit{iter}, \mathit{next}).\ \begin{pmatrix} \lambda y.\ \text{let } r = \mathit{size}(y) \text{ in emit } \texttt{size}; r, \\ \lambda y.\ \mathit{add}(y); \text{emit } \texttt{add}, \\ \lambda y.\ \mathit{remove}(y); \text{emit } \texttt{remove}, \\ \lambda \_.\ \text{let } r = \mathit{iter}() \text{ in emit} \langle \texttt{iterator}, r \rangle; r, \\ \lambda y.\ \text{let } r = \mathit{next}(y) \text{ in emit} \langle \texttt{next}, y \rangle; r \end{pmatrix}$$

The traces ignore the arguments to *add* and *remove* and the return values of *size* and *next*, as they are irrelevant for the protocol.

With this instrumentation we can now express the informal protocol as a language of permissible interaction traces between the client and library. We let the trace alphabet be the countable set:

$$\Sigma = \{\texttt{size}, \texttt{add}, \texttt{remove}\} \cup \{\langle \texttt{next}, \ell \rangle, \langle \texttt{iterator}, \ell \rangle \mid \ell \in \mathit{Loc}\}$$

The language $\mathcal{L}_{\text{coll}}$ of safe behaviours contains all strings $t \in \Sigma^*$ such that, for all $0 \leq i < |t|$:

> if $t[i] = \langle \texttt{next}, \ell \rangle$ then there is $j < i$ such that $t[j] = \langle \texttt{iterator}, \ell \rangle$ and, for all $j < k < i$, $t[k] \notin \{\, \texttt{add}, \texttt{remove} \,\}$.

That is, every call to the *next* method of an iterator $\ell$ must be preceded by a call to *iterator*() which returns $\ell$. In addition, there should be no modification of the collection between those two events.

We now aim to prove that, for an arbitrary library implementation $M$ that satisfies the collections specification, the wrapped library implementation $\mathbf{wrap}_{\mathsf{coll}}(M)$ also satisfies the collections specification *and* the traces generated by the wrapped implementation are in the language $\mathcal{L}_{\mathsf{coll}}$.

Lemma 5.2. $\forall P_0, ops.\ \Phi_{coll}(P_0, ops) \vdash \Phi_{coll}(P_0 * \mathsf{trace}(\varepsilon) * \mathsf{traceInv}(\mathcal{L}_{coll}), \mathbf{wrap}_{coll}(ops))$.

Proof Sketch. We first need to define new wrapped versions of the coll and iter resources that relate the abstract version number to the trace state. The key idea is that the collection parameter of the wrapped coll resource will consist of a pair $(c, n)$, where the $c$ component is the parameter of the underlying coll resource and $n$ is the index in the trace of the last add or remove event. We want the wrapped $\mathsf{coll}((c, n))$ resource to assert that there are no add or remove events in the current trace after the $n$-th element. Likewise, the wrapped $\mathsf{iter}(r, (c, n))$ resource should assert that there *is* an iterator event for iterator $r$ in the current trace after the $n$-th element. Hence, if we own both $\mathsf{coll}((c, n))$ and $\mathsf{iter}(r, (c, n))$ then we know that no add or remove events were emitted since the iterator $r$ was created.

Let coll and iter denote the non-wrapped representation predicates that exist by the $\Phi_{\mathsf{coll}}(P_0, ops)$ assumption and define the wrapped representation predicates $\mathsf{coll}_{\mathsf{w}}$ and $\mathsf{iter}_{\mathsf{w}}$ as follows:

$$\begin{aligned}\mathsf{coll}_{\mathsf{w}}(x) \triangleq\ & \exists y : \mathit{Val}.\, \exists n : \mathbb{N}.\, \lceil x = (y, n) \rceil * \mathsf{coll}(y) * \\ & \exists t.\, \mathsf{trace}(t) * \mathsf{traceInv}(\mathcal{L}_{\mathsf{coll}}) * \lceil \mathsf{add}, \mathsf{remove} \notin t[n..] \wedge n \leq |t| \rceil \\ \mathsf{iter}_{\mathsf{w}}(r, x) \triangleq\ & \exists y : \mathit{Val}.\, \exists n : \mathbb{N}.\, \lceil x = (y, n) \rceil * \mathsf{iter}(y) * \exists t.\, \mathsf{hist}(t) * \lceil \langle \mathtt{iterator}, r \rangle \in t[n..] \rceil\end{aligned}$$

where we use $t[n..]$ as notation for the subtrace of $t$ starting from the $n$-th element.

It thus remains to show that the wrapped library satisfies the collections specification. First, we need to prove that we obtain a wrapped collection resource from the initial resources: $P_0 * \mathsf{traceInv}(\mathcal{L}_{\mathsf{coll}}) * \mathsf{trace}(\varepsilon) \mathrel{-\!\!*} \exists c : \mathit{Val}.\, \mathsf{coll}_{\mathsf{w}}(c)$. This follows easily from the $P_0 \mathrel{-\!\!*} \exists c' : \mathit{Val}.\, \mathsf{coll}(c')$ assumption by taking the second component of $c$ to be 0.

Next, we have to show that each of the wrapped operations satisfies the corresponding Hoare triple. The *size* method is particularly simple, as any trace $t \in \mathcal{L}_{\mathsf{coll}}$ can trivially be extended with a size event $t \cdot \mathtt{size} \in \mathcal{L}_{\mathsf{coll}}$. We will thus skip the *size* method. The *add* method is more interesting, as we have to update the index into the trace for the last *add* event. Below is a proof outline for the wrapped *add* method applied to argument $z$.

---

$\{\mathsf{coll}_{\mathsf{w}}(c)\}$
$\{\exists x, n, t.\, \lceil c = (x, n) \rceil * \mathsf{coll}(x) * \mathsf{trace}(t) * \mathsf{traceInv}(\mathcal{L}_{\mathsf{coll}}) * \lceil \mathsf{add}, \mathsf{remove} \notin t[n..] \wedge n \leq |t| \rceil\}$
  add(z);
$\{\exists x', n, t.\, \mathsf{coll}(x') * \mathsf{trace}(t) * \mathsf{traceInv}(\mathcal{L}_{\mathsf{coll}}) * \lceil \mathsf{add}, \mathsf{remove} \notin t[n..] \wedge n \leq |t| \rceil\}$
  **emit** add;
$\{\exists x', n, t.\, \mathsf{coll}(x') * \mathsf{trace}(t \cdot \mathsf{add}) * \mathsf{traceInv}(\mathcal{L}_{\mathsf{coll}}) * \lceil \mathsf{add}, \mathsf{remove} \notin t[n..] \wedge n \leq |t| \rceil\}$
$\{\exists c' : \mathit{Val}.\, \mathsf{coll}_{\mathsf{w}}(c')\}$

---

This leaves us with two proof obligations: firstly, we have to show that we are allowed to emit the add event (i.e., that $t \cdot \mathsf{add} \in \mathcal{L}_{\mathsf{coll}}$); and secondly for the last step we have to show that:

$$\forall x', n, t. \begin{pmatrix} \mathsf{coll}(x') * \mathsf{trace}(t \cdot \mathsf{add}) * \mathsf{traceInv}(\mathcal{L}_{\mathsf{coll}}) \\ * \lceil \mathsf{add}, \mathsf{remove} \notin t[n..] \wedge n \leq |t| \rceil \end{pmatrix} \mathrel{-\!\!*} \exists c' : \mathit{Val}.\, \mathsf{coll}_{\mathsf{w}}(c')$$

This follows easily by taking $c'$ to be $(x', |t \cdot \mathsf{add}|)$, as $t \cdot \mathsf{add}$ contains no add or remove events after the $|t \cdot \mathsf{add}|$-th element. The proof for *remove* follows the same structure as for *add*.

For the *iterator* method, we emit an `iterator` event and create a new hist resource to record the trace at the time of the creation of the iterator. One the right, we give a proof outline for the *iterator* method. We are left with two proof obligations: $t \cdot \langle \texttt{iterator}, \mathsf{r} \rangle \in \mathcal{L}_{\text{coll}}$, and:

$$\begin{aligned}&\forall x, n, t.\\ &\quad (\lceil c = (x, n) \rceil * \mathsf{coll}(x) * \mathsf{iter}(\mathsf{r}, x) *\\ &\quad\ \mathsf{trace}(t \cdot \langle \texttt{iterator}, \mathsf{r} \rangle) * \mathsf{traceInv}(\mathcal{L}_{\text{coll}}) *\\ &\quad\ \lceil \mathsf{add}, \mathsf{remove} \notin t[n..] \wedge n \leq |t| \rceil)\\ &\quad -\!\!* \ \mathsf{coll}_{\mathsf{w}}(c) * \mathsf{iter}_{\mathsf{w}}(\mathsf{r}, c)\end{aligned}$$

$\{\mathsf{coll}_{\mathsf{w}}(c)\}$
$\{\exists x, n, t.\ \lceil c = (x, n) \rceil * \mathsf{coll}(x) * \mathsf{trace}(t) *$
$\quad \mathsf{traceInv}(\mathcal{L}_{\text{coll}}) * \lceil \mathsf{add}, \mathsf{remove} \notin t[n..] \wedge n \leq |t| \rceil\}$
$\quad$ **let** r = iterator() **in**
$\{\exists x, n, t.\ \lceil c = (x, n) \rceil * \mathsf{coll}(x) * \mathsf{iter}(\mathsf{r}, x) * \mathsf{trace}(t) *$
$\quad \mathsf{traceInv}(\mathcal{L}_{\text{coll}}) * \lceil \mathsf{add}, \mathsf{remove} \notin t[n..] \wedge n \leq |t| \rceil\}$
$\quad$ **emit**⟨iterator, r⟩;
$\{\exists x, n, t.\ \lceil c = (x, n) \rceil * \mathsf{coll}(x) * \mathsf{iter}(\mathsf{r}, x) *$
$\quad \mathsf{trace}(t \cdot \langle \texttt{iterator}, \mathsf{r} \rangle) * \mathsf{traceInv}(\mathcal{L}_{\text{coll}}) *$
$\quad \lceil \mathsf{add}, \mathsf{remove} \notin t[n..] \wedge n \leq |t| \rceil\}$
$\{\mathsf{coll}_{\mathsf{w}}(c) * \mathsf{iter}_{\mathsf{w}}(\mathsf{r}, c)\}$
$\quad$ r
$\{r.\ \mathsf{coll}_{\mathsf{w}}(c) * \mathsf{iter}_{\mathsf{w}}(r, c)\}$

To discharge this last proof obligation, we use the rule from Figure 3 that allows us to introduce a history resource $\mathsf{hist}(t \cdot \langle \texttt{iterator}, \mathsf{r} \rangle)$ and since $n \leq |t|$ it follows that $\langle \texttt{iterator}, \mathsf{r} \rangle \in (t \cdot \langle \texttt{iterator}, \mathsf{r} \rangle)[n..]$, as required by $\mathsf{iter}_{\mathsf{w}}(\mathsf{r}, c)$.

We are left with `next`, which is the most interesting case as it requires us to prove the iterator we are trying to use is still valid. We give a proof outline for the `next` method applied to a value x:

$\{\mathsf{coll}_{\mathsf{w}}(c) * \mathsf{iter}_{\mathsf{w}}(\mathsf{x}, c)\}$
$\{\exists y, n, t, t'.\ \lceil c = (y, n) \rceil * \mathsf{coll}(y) * \mathsf{iter}(\mathsf{x}, c) * \mathsf{trace}(t) * \mathsf{hist}(t') * \mathsf{traceInv}(\mathcal{L}_{\text{coll}})$
$\quad * \lceil \mathsf{add}, \mathsf{remove} \notin t[n..] \wedge n \leq |t| \rceil * \lceil \langle \texttt{iterator}, \mathsf{x} \rangle \in t'[n..] \rceil\}$
$\quad$ **let** r = next(x);
$\{\exists y, n, t, t'.\ \lceil c = (y, n) \rceil * \mathsf{coll}(y) * \mathsf{iter}(\mathsf{x}, c) * \mathsf{trace}(t) * \mathsf{hist}(t') * \mathsf{traceInv}(\mathcal{L}_{\text{coll}})$
$\quad * \lceil \mathsf{add}, \mathsf{remove} \notin t[n..] \wedge n \leq |t| \rceil * \lceil \langle \texttt{iterator}, \mathsf{x} \rangle \in t'[n..] \rceil\}$
$\quad$ **emit**⟨next, x⟩;
$\{\exists y, n, t, t'.\ \lceil c = (y, n) \rceil * \mathsf{coll}(y) * \mathsf{iter}(\mathsf{x}, c) * \mathsf{trace}(t \cdot \langle \texttt{next}, \mathsf{x} \rangle) * \mathsf{hist}(t') * \mathsf{traceInv}(\mathcal{L}_{\text{coll}})$
$\quad * \lceil \mathsf{add}, \mathsf{remove} \notin t[n..] \wedge n \leq |t| \rceil * \lceil \langle \texttt{iterator}, \mathsf{x} \rangle \in t'[n..] \rceil\}$
$\{\mathsf{coll}_{\mathsf{w}}(c) * \mathsf{iter}_{\mathsf{w}}(\mathsf{x}, c)\}$
$\quad$ r
$\{\mathsf{coll}_{\mathsf{w}}(c) * \mathsf{iter}_{\mathsf{w}}(\mathsf{x}, c)\}$

To verify the emit expression, we further have to prove that $t \cdot \langle \texttt{next}, \mathsf{x} \rangle \in \mathcal{L}_{\text{coll}}$, that is, that the iterator x is still valid. This relies on the following key property of the $\mathcal{L}_{\text{coll}}$ language:

$$t \in \mathcal{L}_{\text{coll}} \wedge \mathsf{add}, \mathsf{remove} \notin t[n..] \wedge \langle \texttt{iterator}, \mathsf{x} \rangle \in t[n..] \implies t \cdot \langle \texttt{next}, \mathsf{x} \rangle \in \mathcal{L}_{\text{coll}}$$

To apply this property we use the adequate rule from Figure 3 to conclude from $\mathsf{trace}(t) * \mathsf{hist}(t')$ that $t' \leq_{pref} t$ and thus that $\langle \texttt{iterator}, \mathsf{x} \rangle \in t'[n..] \implies \langle \texttt{iterator}, \mathsf{x} \rangle \in t[n..]$.

### 5.3 Well-bracketing Protocols

Libraries that allow clients to acquire, access and release resources often impose a well-bracketing protocol whereby clients are required to acquire resources before accessing and releasing them. The file library in §5.1 was a particularly simple example of such a protocol. In this section we consider a more advanced and realistic variant thereof for a library with a higher-order function that takes care of acquiring and releasing the underlying resource for clients.

Consider a library with a higher-order method *withRes* for acquiring, accessing and subsequently releasing some resource (e.g. a file) and an operation *op* for accessing the resource. The *withRes* operation takes as argument a function $f$ provided by the client for accessing the resource and takes care of acquiring the resource before $f$ is called and subsequently releasing it again. For such a library, we typically wish to make sure that (1) clients only access the resource after they have

acquired it, and (2) clients do not try to acquire resources they already hold. Such a library can be given a specification in Separation Logic as follows:

$$\Phi_{\text{brac}}(P_0, (\mathit{withRes}, \mathit{op})) \triangleq$$
$$\exists \text{locked} : \mathit{iProp}.\, \exists \text{unlocked} : \mathit{Val} \to \mathit{iProp}.\ \square(P_0 \mathrel{-\!\!*} \text{locked}) \wedge$$
$$\forall P, Q : \mathit{iProp}.\, \forall f : \mathit{Val}.\, \{\text{locked} * P * S(P, Q, \text{unlocked}, f)\}\ \mathit{withRes}(f)\ \{\text{locked} * Q\} \wedge$$
$$\forall x, y : \mathit{Val}.\, \{\text{unlocked}(y)\}\ \mathit{op}(x)\ \{\text{unlocked}(y)\}$$

$$S(P, Q, \text{unlocked}, f) \triangleq \forall x, y.\, \{\text{unlocked}(y) * P\}\ f(x)\ \{\text{unlocked}(y) * Q\}$$

This specification relies on two abstract resources, unlocked and locked, to capture the well-bracketing aspect of the protocol. One can see that calling *withRes* requires the client to relinquish ownership of the locked resource. Since the function provided by the client is only given ownership of the abstract unlocked resource, it cannot make reentrant calls to *withRes*. Furthermore, to call *op* requires ownership of the unlocked resource, thus ensuring that only the callback provided by the client to *withRes* can call *op*.

But this specification entails another less obvious property. In the specification of *withRes*, the library must behave parametrically with respect to $P$ and $Q$, which are provided by the client. Thus, the specification ensures that *withRes* is forced to call the function provided by the client *exactly once*, as the only way it can transform $P$ into $Q$ is by calling the function provided by the client.

As before, we wish to establish these properties as free theorems obtained from the specification. We define a wrapping function that instruments the library to emit call and return events for all calls where control passes between client and library.

$$\begin{aligned}\mathbf{wrap}_{\text{brac}} \triangleq \lambda(\mathit{withRes}, \mathit{op}).\, (\ &\lambda f.\, \mathsf{emit}\langle \mathtt{call}, \mathtt{withRes}, f\rangle;\\ &\mathit{withRes}(\lambda x.\, \mathsf{emit}\langle \mathtt{call}, f\rangle; f(x); \mathsf{emit}\langle \mathtt{ret}, f\rangle);\\ &\mathsf{emit}\langle \mathtt{ret}, \mathtt{withRes}, f\rangle,\\ &\lambda \mathtt{x}.\, \mathsf{emit}\langle \mathtt{call}, \mathtt{op}\rangle; \mathit{op}(\mathtt{x}); \mathsf{emit}\langle \mathtt{ret}, \mathtt{op}\rangle\ )\end{aligned}$$

We now state formally the desired property as a well-bracketing property of the traces generated by the instrumented library. Let us define $\mathcal{L}_{\text{bracfull}} \subseteq \mathit{Val}^*$ the language of traces of the form:

$$\langle \mathtt{call}, \mathtt{withRes}, f\rangle \cdot \langle \mathtt{call}, f\rangle \cdot s_{\text{op}} \cdot \langle \mathtt{ret}, f\rangle \cdot \langle \mathtt{ret}, \mathtt{withRes}, f\rangle \cdot s'$$

for some $f \in \mathit{Val}$, $s_{\text{op}} \in (\langle \mathtt{call}, \mathtt{op}\rangle \cdot \langle \mathtt{ret}, \mathtt{op}\rangle)^*$ and $s' \in \mathcal{L}_{\text{bracfull}}$. That is, the traces in $\mathcal{L}_{\text{bracfull}}$ are well-bracketed sequences of events formed of subsequences adhering to the pattern $\langle \mathtt{call}, \mathtt{withRes}, f\rangle \cdot \langle \mathtt{call}, f\rangle \cdot s_{\text{op}} \cdot \langle \mathtt{ret}, f\rangle \cdot \langle \mathtt{ret}, \mathtt{withRes}, f\rangle$, and subsequences thereof, where $s_{\text{op}}$ a sequence of consecutive calls and returns of op. Then, we take $\mathcal{L}_{\text{brac}}$, the language of traces we wish to enforce, to be the prefix closure of $\mathcal{L}_{\text{bracfull}}$.

To prove that the specification enforces the trace property we proceed as usual, by proving that for any implementation $M$ that satisfies the specification, the wrapped implementation $\mathbf{wrap}_{\text{brac}}(M)$ also satisfies the specification and generates traces in $\mathcal{L}_{\text{brac}}$.

Lemma 5.3. $\forall P_0, \mathit{ops}.\ \Phi_{\mathit{brac}}(P_0, \mathit{ops}) \vdash \Phi_{\mathit{brac}}(P_0 * \mathsf{trace}(\varepsilon) * \mathsf{traceInv}(\mathcal{L}_{\text{brac}}), \mathbf{wrap}_{\mathit{brac}}(\mathit{ops}))$.

Proof Sketch. To prove that the wrapped version satisfies the specification and produces traces in $\mathcal{L}_{\text{brac}}$, we first need to define wrapped versions of the abstract representation predicates. The idea is to let the wrapped locked resource, $\text{locked}_{\text{w}}$, express that the current trace $t$ is in $\mathcal{L}_{\text{brac}}$ and the $\langle \mathtt{call}, \mathtt{withRes}, f\rangle$ and $\langle \mathtt{ret}, \mathtt{withRes}, f\rangle$ events in $t$ are well-balanced and well-bracketed. For the wrapped unlocked resource, $\text{unlocked}_{\text{w}}(x)$, the idea is to use the argument $x$ to track the name $f$ of the last unbalanced $\langle \mathtt{call}, \mathtt{withRes}, f\rangle$ event in $t$.

To simplify the definitions and subsequent proofs, we first introduce a number of auxiliary resources, $T_0, T_1, T_2$ and $T_3$. $T_0$ expresses that the current trace is well-balanced. We set $O = (\langle \mathtt{call}, \mathsf{op}\rangle \cdot \langle \mathtt{ret}, \mathsf{op}\rangle)^*$. $T_1(f)$ expresses that the current trace $t$ has the form $s \cdot \langle \mathtt{call}, \mathtt{withRes}, f\rangle$ where $s$ is well-balanced. $T_2(f)$ expresses that the current trace $t$ has the form $s \cdot \langle \mathtt{call}, \mathtt{withRes}, f\rangle \cdot \langle \mathtt{call}, f\rangle \cdot s'$ where $s$ is well-balanced and $s' \in O$. Finally, $T_3(f)$ expresses that the current trace $t$ has the form $s \cdot \langle \mathtt{call}, \mathtt{withRes}, f\rangle \cdot \langle \mathtt{call}, f\rangle \cdot s' \cdot \langle \mathtt{ret}, f\rangle$ where $s$ is well-balanced and $s' \in O$.

$$
\begin{aligned}
T_0 &= \exists t.\, \mathsf{trace}(t) * \mathsf{traceInv}(\mathcal{L}_{\mathsf{brac}}) * \lceil t \in \mathcal{L}_{\mathsf{bracfull}} \rceil \\
T_1(f) &= \exists t.\, \mathsf{trace}(t) * \mathsf{traceInv}(\mathcal{L}_{\mathsf{brac}}) * \exists t' \in \mathcal{L}_{\mathsf{bracfull}}.\, \lceil t = t' \cdot \langle \mathtt{call}, \mathtt{withRes}, f\rangle \rceil \\
T_2(f) &= \exists t.\, \mathsf{trace}(t) * \mathsf{traceInv}(\mathcal{L}_{\mathsf{brac}}) * \\
&\qquad \exists t' \in \mathcal{L}_{\mathsf{bracfull}}, s_{\mathsf{op}} \in O.\, \lceil t = t' \cdot \langle \mathtt{call}, \mathtt{withRes}, f\rangle \cdot \langle \mathtt{call}, f\rangle \cdot s_{\mathsf{op}} \rceil \\
T_3(f) &= \exists t.\, \mathsf{trace}(t) * \mathsf{traceInv}(\mathcal{L}_{\mathsf{brac}}) * \\
&\qquad \exists t' \in \mathcal{L}_{\mathsf{bracfull}}, s_{\mathsf{op}} \in O.\, \lceil t = t' \cdot \langle \mathtt{call}, \mathtt{withRes}, f\rangle \cdot \langle \mathtt{call}, f\rangle \cdot s_{\mathsf{op}} \cdot \langle \mathtt{ret}, f\rangle \rceil
\end{aligned}
$$

With these resources, we can now define $\mathsf{unlocked}_{\mathsf{w}}$ and $\mathsf{locked}_{\mathsf{w}}$:

$$
\begin{aligned}
\mathsf{unlocked}_{\mathsf{w}}(x) &\triangleq \exists y, z : \mathit{Val}.\, \lceil x = (y, z) \rceil * \mathsf{unlocked}(y) * T_2(z) \\
\mathsf{locked}_{\mathsf{w}} &\triangleq \mathsf{locked} * T_0
\end{aligned}
$$

It follows easily that $P_0 * \mathsf{trace}(\varepsilon) * \mathsf{traceInv}(\mathcal{L}_{\mathsf{brac}}) \mathrel{-\!\!*} \mathsf{locked}_{\mathsf{w}}$ from $\mathsf{trace}(\varepsilon) * \mathsf{traceInv}(\mathcal{L}_{\mathsf{brac}}) \mathrel{-\!\!*} T_0$ and the assumption $P_0 \mathrel{-\!\!*} \mathsf{locked}$.

It remains to show that the two instrumented operations satisfy their specifications. We begin by showing that the instrumented *withRes* operation satisfies its specification:

$$
\begin{aligned}
&\forall P, Q : \mathit{iProp}.\, \forall f : \mathit{Val}. \\
&\{\mathsf{locked}_{\mathsf{w}} * P * S(P, Q, \mathsf{unlocked}_{\mathsf{w}}, f)\}\ \mathsf{fst}\ (\mathbf{wrap}_{\mathsf{brac}}(\mathit{withRes}, \mathit{op}))(f)\ \{\mathsf{locked}_{\mathsf{w}} * Q\}
\end{aligned}
$$

assuming *withRes* satisfies its specification:

$$
\forall P, Q : \mathit{iProp}.\, \forall f : \mathit{Val}.\, \{\mathsf{locked} * P * S(P, Q, \mathsf{unlocked}, f)\}\ \mathit{withRes}(f)\ \{\mathsf{locked} * Q\}.
$$

We give a proof outline for the instrumented *withRes* operation:

$\{\mathsf{locked}_{\mathsf{w}} * P * S(P, Q, \mathsf{unlocked}_{\mathsf{w}}, f)\}$
$\{\mathsf{locked} * T_0 * P * S(P, Q, \mathsf{unlocked}_{\mathsf{w}}, f)\}$
  **emit**⟨call, withRes, f⟩;
$\{\mathsf{locked} * T_1(f) * P * S(P, Q, \mathsf{unlocked}_{\mathsf{w}}, f)\}$
  **let** g = λx. **emit**⟨call, f⟩; f(x); **emit**⟨ret, f⟩ **in**
$\{\mathsf{locked} * T_1(f) * P *$
$\quad S(P * T_1(f), Q * T_3(f), \mathsf{unlocked}, g)\}$
  withRes(g);
$\{\mathsf{locked} * T_3(f) * Q\}$
  **emit**⟨ret, withRes, f⟩
$\{\mathsf{locked} * T_0 * Q\}$
$\{\mathsf{locked}_{\mathsf{w}} * Q\}$

$\{\mathsf{unlocked}(y) * P * T_1(f) * S(P, Q, \mathsf{unlocked}_{\mathsf{w}}, f)\}$
  **emit**⟨call, f⟩;
$\{\mathsf{unlocked}(y) * P * T_2(f) * S(P, Q, \mathsf{unlocked}_{\mathsf{w}}, f)\}$
$\{\mathsf{unlocked}_{\mathsf{w}}((y, f)) * P * S(P, Q, \mathsf{unlocked}_{\mathsf{w}}, f)\}$
  f(x);
$\{\mathsf{unlocked}_{\mathsf{w}}((y, f)) * Q\}$
$\{\mathsf{unlocked}(y) * Q * T_2(f)\}$
  **emit**⟨ret, f⟩
$\{\mathsf{unlocked}(y) * Q * T_3(f)\}$

The interesting step, that we detail on the right-hand side of the proof outline, is showing that from the assumed specification of $f$ we can derive the desired specification for the instrumented version of $f$:

$$
\begin{aligned}
&\forall P, Q : \mathit{iProp}.\, \forall f : \mathit{Val}.\ S(P, Q, \mathsf{unlocked}_{\mathsf{w}}, f) \Rightarrow \\
&\qquad\qquad S(P * T_1(f), Q * T_3(f), \mathsf{unlocked}, \lambda x.\ \mathsf{emit}\langle \mathtt{call}, f\rangle; f(x); \mathsf{emit}\langle \mathtt{ret}, f\rangle).
\end{aligned}
$$

Lastly, we need to show that the instrumented *op* function satisfies its specification. This follows easily from the fact that $\mathsf{unlocked}_w$ enforces $T_2(f)$, which is itself preserved when emitting a new $\langle \mathtt{call}, \mathsf{op} \rangle$ event. □

### 5.4 Traversable stack example

To further demonstrate that our approach can express and enforce strong trace properties, recall the stack example from §2. We have a stack with a *push* and a pop method, and a *foreach* method that takes a function argument and applies the given function to every element of the stack, in order, starting from the top-most element. Here the protocol on the interaction between client and library imposes restrictions on both the client and the library. In particular, we wish to ensure that the function provided by the client cannot call back into the stack-library and potentially modify the underlying stack during the iteration of the stack. We also wish to ensure that the library calls the function provided by the client with every element currently on the stack and in the right order.

A higher-order separation logic specification for such a stack data structure is the following.

$$
\begin{aligned}
&\Phi(P_{init}, (push, pop, foreach)) \triangleq \\
&\quad \exists \mathsf{stack} : \mathit{Val}^* \to \mathit{iProp}.\ \Box(P_{init} \mathrel{-\!\!*} \mathsf{stack}(\varepsilon)) \wedge \\
&\quad \forall \alpha, x.\ \{\mathsf{stack}(\alpha) * \lceil x \neq ()\rceil\}\ push(x)\ \{\mathsf{stack}(x :: \alpha)\} \wedge \\
&\quad \forall \alpha.\ \{\mathsf{stack}(\alpha)\}\ pop()\ \{r.\ (\lceil r = () \wedge \alpha = \varepsilon\rceil * \mathsf{stack}(\alpha)) \vee (\exists \alpha'.\ \lceil \alpha = r :: \alpha'\rceil \wedge \mathsf{stack}(\alpha'))\} \wedge \\
&\quad \forall \alpha, f, I.\ \{\mathsf{stack}(\alpha) * I(\varepsilon) * \forall \beta, x.\ \{I(\beta)\}\ f(x)\ \{I(x :: \beta)\}\}\ foreach(f)\ \{\mathsf{stack}(\alpha) * I(rev(\alpha))\}
\end{aligned}
$$

It asserts existence of an abstract stack representation predicate $\mathsf{stack}(\alpha)$ that tracks the exact sequence of elements currently on the stack using the mathematical sequence $\alpha$. The specification for *push* and *pop* is straightforward: pushing and popping elements pushes or pops elements from this mathematical sequence, with a few special cases for pushing () or popping from an empty stack. The specification for *foreach* is more interesting. It is parametrised by a predicate $I$, to be chosen by the client. This predicate is indexed by a sequence $\beta$ and $I(\beta)$ is intended to capture the client's state after the function provided by the client has been called on each element of $\beta$, in reverse order. This accounts for the $I(rev(\beta))$ in the post-condition, where *rev* is the reverse operator on sequences.

We now wish to show as a free theorem that any implementation satisfying this Separation Logic stack specification will satisfy the informal trace property described above. We can formalize the intended protocol as the language $\mathcal{L}_{\mathsf{stack}}$ defined as the prefix closure of the language of all traces $t$ such that $\mathsf{stk}_{\mathsf{tr}}(t, \varepsilon)$ holds, where, given $\alpha \in \mathit{Val}^*$, we define $\mathsf{stk}_{\mathsf{tr}}(t, \alpha)$ by:

$$
\begin{aligned}
&\mathsf{stk}_{\mathsf{tr}}(t, \alpha) \triangleq (t = \alpha = \varepsilon) \vee \\
&\quad (t = t' \cdot \langle \mathtt{call}, \mathsf{push}, x\rangle \cdot \langle \mathtt{ret}, \mathsf{push}\rangle \wedge \alpha = x :: \alpha' \wedge \mathsf{stk}_{\mathsf{tr}}(t', \alpha')) \vee \\
&\quad (t = t' \cdot \langle \mathtt{call}, \mathsf{pop}\rangle \cdot \langle \mathtt{ret}, \mathsf{pop}, ()\rangle \wedge \alpha = \varepsilon \wedge \mathsf{stk}_{\mathsf{tr}}(t', \varepsilon)) \vee \\
&\quad (t = t' \cdot \langle \mathtt{call}, \mathsf{pop}\rangle \cdot \langle \mathtt{ret}, \mathsf{pop}, x\rangle \wedge \mathsf{stk}_{\mathsf{tr}}(t', x :: \alpha)) \vee \\
&\quad (t = t' \cdot \langle \mathtt{call}, \mathtt{foreach}, f\rangle \cdot t'' \cdot \langle \mathtt{ret}, \mathtt{foreach}\rangle \wedge \mathsf{stk}_{\mathsf{tr}}(t', \alpha) \wedge \mathsf{trav}(t'', \alpha, f)) \\
&\mathsf{trav}(t, \alpha, f) \triangleq (t = \alpha = \varepsilon) \vee (t = \langle \mathtt{call}, f, x\rangle \cdot \langle \mathtt{ret}, f\rangle \cdot t' \wedge \alpha = x :: \alpha' \wedge \mathsf{trav}(t', \alpha', f))
\end{aligned}
$$

To prove the free theorem we then define a suitable library wrapper that tracks all calls to push and pop and all calls to the function argument provided by the client when calling foreach.

$$\begin{aligned}&\mathbf{wrap}_{\mathsf{stack}}(\mathit{push}, \mathit{pop}, \mathit{foreach}) \triangleq\\ &(\lambda x.\ \mathsf{emit}\langle \mathtt{call}, \mathsf{push}, x\rangle; \mathit{push}(x); \mathsf{emit}\langle \mathtt{ret}, \mathsf{push}\rangle,\\ &\ \lambda\_.\ \mathsf{emit}\langle \mathtt{call}, \mathsf{pop}\rangle; \mathsf{let}\ x = \mathit{pop}()\ \mathsf{in}\ \mathsf{emit}\langle \mathtt{ret}, \mathsf{pop}, x\rangle; x,\\ &\ \lambda f.\ \mathsf{emit}\langle \mathtt{call}, \mathtt{foreach}, f\rangle; \mathit{foreach}\big(\lambda x.\ \mathsf{emit}\langle \mathtt{call}, f, x\rangle; f(x);\\ &\qquad \mathsf{emit}\langle \mathtt{ret}, f\rangle\big); \mathsf{emit}\langle \mathtt{ret}, \mathtt{foreach}\rangle)\end{aligned}$$

Lemma 5.4. $\forall P_0, \mathit{ops}.\ \Phi(P_0, \mathit{ops}) \vdash \Phi(P_0 * \mathsf{traceInv}(\mathcal{L}_{\mathsf{stack}}) * \mathsf{trace}(\varepsilon), \mathbf{wrap}_{\mathit{stack}}(\mathit{ops}))$.

Proof Sketch. We proceed by defining a wrapped version of the stack predicate that asserts that the sequence of elements $\alpha$ matches the contents of the stack as per the current trace $t$.

$$\mathsf{stack}_{\mathsf{w}}(\alpha) \triangleq \mathsf{stack}(\alpha) * \exists t.\ \lceil \mathsf{stk}_{\mathsf{tr}}(t, \alpha)\rceil * \mathsf{trace}(t) * \mathsf{traceInv}(\mathcal{L}_{\mathsf{stack}})$$

Clearly we have that $P_{init} * \mathsf{traceInv}(\mathcal{L}_{\mathsf{stack}}) * \mathsf{trace}(\varepsilon) \mathrel{-\!\!*} \mathsf{stack}_{\mathsf{w}}(\varepsilon)$ and $P_{init} \mathrel{-\!\!*} \mathsf{stack}(\varepsilon)$.

It remains to prove the wrapped library methods satisfy the specification instantiated with the wrapped stack predicate. The proofs for *push* and *pop* are straightforward and have been omitted. For *foreach* we are given a predicate $I$ from the client and need to prove the following triple:

$$\begin{aligned}&\{\mathsf{stack}_{\mathsf{w}}(\alpha) * I(\varepsilon) * \forall \beta, x.\ \{I(\beta)\}\ f(x)\ \{I(x :: \beta)\}\}\\ &\quad (\mathsf{snd}\ (\mathsf{snd}\ (\mathbf{wrap}_{\mathsf{stack}}(\mathit{ops})))(f)\\ &\{\mathsf{stack}_{\mathsf{w}}(\alpha) * I(\mathit{rev}(\alpha)))\}\end{aligned}$$

In the call to the underlying *foreach* method, we can pick a suitably wrapped version of the $I$ predicate, $I_{\mathsf{w}}(\beta)$. The idea is that it should assert $I(\beta)$ and that we have emitted an opening *foreach* call and called the function argument on all the elements of $\beta$ so far.

$$I_{\mathsf{w}}(\beta) \triangleq I(\beta) * \exists t_1, t_2.\ \lceil \mathsf{stk}_{\mathsf{tr}}(t_1, \alpha) \wedge \mathsf{trav}(t_2, \mathit{rev}(\beta), f)\rceil * \mathsf{trace}(t_1 \cdot \langle \mathtt{call}, \mathtt{foreach}, f\rangle \cdot t_2)$$

We need to prove that the wrapped function argument updates the wrapped $I$ predicate appropriately. This follows from:

$\{I_{\mathsf{w}}(\beta)\}$
$\{I(\beta) * \exists t_1, t_2.\ \lceil \mathsf{stk}_{\mathsf{tr}}(t_1, \alpha) \wedge \mathsf{trav}(t_2, \mathit{rev}(\beta), f)\rceil * \mathsf{trace}(t_1 \cdot \langle \mathtt{call}, \mathtt{foreach}, f\rangle \cdot t_2)\}$
  **emit** ⟨call, $f$, $x$⟩;
$\{I(\beta) * \exists t_1, t_2.\ \lceil \mathsf{stk}_{\mathsf{tr}}(t_1, \alpha) \wedge \mathsf{trav}(t_2, \mathit{rev}(\beta), f)\rceil * \mathsf{trace}(t_1 \cdot \langle \mathtt{call}, \mathtt{foreach}, f\rangle \cdot t_2 \cdot \langle \mathtt{call}, f, x\rangle)\}$
  f(x);
$\{I(x :: \beta) * \exists t_1, t_2.\ \lceil \mathsf{stk}_{\mathsf{tr}}(t_1, \alpha) \wedge \mathsf{trav}(t_2, \mathit{rev}(\beta), f)\rceil * \mathsf{trace}(t_1 \cdot \langle \mathtt{call}, \mathtt{foreach}, f\rangle \cdot t_2 \cdot \langle \mathtt{call}, f, x\rangle)\}$
  **emit** ⟨ret, $f$⟩;
$\{I(x :: \beta) * \exists t_1, t_2.\ \lceil \mathsf{stk}_{\mathsf{tr}}(t_1, \alpha) \wedge \mathsf{trav}(t_2, \mathit{rev}(\beta), f)\rceil$
    $* \mathsf{trace}(t_1 \cdot \langle \mathtt{call}, \mathtt{foreach}, f\rangle \cdot t_2 \cdot \langle \mathtt{call}, f, x\rangle \cdot \langle \mathtt{ret}, f\rangle)\}$
$\{I(x :: \beta) * \exists t_1, t_2.\ \lceil \mathsf{stk}_{\mathsf{tr}}(t_1, \alpha) \wedge \mathsf{trav}(t_2, \mathit{rev}(x :: \beta), f)\rceil * \mathsf{trace}(t_1 \cdot \langle \mathtt{call}, \mathtt{foreach}, f\rangle \cdot t_2)\}$
$\{I_{\mathsf{w}}(x :: \beta)\}$

The second to last step follows from the following property:

$$\forall \alpha, x, t.\ \mathsf{trav}(t, \alpha, f) \implies \mathsf{trav}(t \cdot \langle \mathtt{call}, f, x\rangle \cdot \langle \mathtt{ret}, f\rangle, \alpha \cdot x, f)$$

Now the rest of the proof of `foreach` is just an application of the specification of the underlying `foreach` method and the Emit rule for the emission of the `foreach` call and return events. □

## 6 RELATING LOGICAL ATOMICITY AND LINEARIZABILITY

The specifications that we have considered so far are sequential in nature. Indeed, even though we are working in a concurrent language, and even though both the implementation of a library and its client can in principle make use of concurrency primitives, the specifications presented earlier enforce that a client may only call the operations of the library in a sequential fashion.

For instance, the specification for the *push* operation of the stack library requires the client to give up exclusive ownership over the stack data structure for the duration of the call, thus preventing concurrent operations to be made.

And indeed, nothing requires an implementation of this stack library to be thread-safe: during the execution of *push*, the internal invariants on the stack data structure might be temporarily broken, in a way that would be observable by concurrent operations. An unverified client attempting to use the library in a concurrent fashion might then observe bogus results or corrupt the data structure.

Proper concurrent data structures are instead carefully designed to ensure that all operations are safe to call concurrently. In particular, they usually ensure that operations appear to take effect atomically, in order to alleviate the need for clients to reason about interleavings of the individual instructions within each operation.

Classically this idea of atomicity is formalized as a property of traces of the interactions between library and client called *linearizability* [Herlihy and Wing 1990]. More recently, in the setting of program logics such as TaDa [da Rocha Pinto et al. 2014] and Iris [Jung et al. 2015a] (see also [Birkedal and Bizjak 2020, Sec. 13]), an alternative notion called *logical atomicity* has been proposed as an alternative to linearizability more amenable to program verification in Hoare logic. Logical atomicity allows expressing the idea of linearizability as an abstract specification for a library, thus allowing clients to reason about operations that appear atomic using the same reasoning principles as physically atomic operations.

In this section, we show how to establish a formal connection between linearizability and logical atomicity. More precisely, we show that any library that can be verified against a logically atomic specification is in fact linearizable.

### 6.1 Logical atomicity

In this subsection we recall how logical atomicity is captured in Iris. As mentioned above, a logically atomic specification for an operation is intended to express that the operation behaves *as if it was atomic*. Now, from a logical perspective, what reasoning power do we gain from this property, when reasoning about calls to logically atomic operations? The answer is that logical atomicity *allows us to access invariants* around logically atomic operations. In Iris, invariants are the key mechanism used to manage resources that are shared and updated between concurrent threads according to certain protocols. Invariants (which have to hold at every step of the execution) can only be accessed for the duration of a single atomic execution step, during which the resources held by the invariant become accessible. This is described by the invariant opening rule INV-OPEN:[3]

$$\textsc{inv-open}\ \frac{\{P * R\}\ e\ \{v.\, Q * R\} \qquad e \text{ is atomic}}{\boxed{R} \vdash \{P\}\ e\ \{v.\, Q\}}$$

What logical atomicity grants us is the power to use a similar invariant opening rule around the call to a logically atomic operation, even though its implementation is not physically atomic (it performs more than one step of computation), as long as it behaves as if it were.

*Logically atomic triples.* Logically atomic operations are specified using logically atomic Hoare triples, $\langle P \rangle\ e\ \langle v.\, Q \rangle$. These triples support an invariant opening rule similar to INV-OPEN:

$$\textsc{logatom-inv-open}\ \frac{\langle P * R \rangle\ e\ \langle v.\, Q * R \rangle}{\boxed{R} \vdash \langle P \rangle\ e\ \langle v.\, Q \rangle}$$

[3] We omit the technical details related to namespaces and masks throughout this section. They are important for soundness but not for the presentation here, and we refer the reader to the Coq formalization for further details.

Note that in contrast to INV-OPEN, this rule does not require the expression $e$ to be physically atomic. Unlike a normal Hoare triple, which expresses that the resources in precondition $P$ are updated to $Q$ after possibly many execution steps by running $e$, a logically atomic triple requires that updates occur during a single execution step. As such, a triple $\langle P \rangle\ e\ \langle v.\, Q \rangle$ is perhaps best seen as the specification for the *linearization point* that occurs during the execution of $e$: the single internal step that atomically updates the observable state of the data structure.

*Using and proving logically atomic triples.* Logically atomic triples are defined in Iris in terms of a more primitive notion of *atomic update*. We have the following rule:

$$\text{LOGATOM-AU}\quad \langle P \rangle\ e\ \langle v.\, Q \rangle \dashv\vdash \forall \Psi.\, \{\mathsf{AU}_{P,Q}(\Psi)\}\ e\ \{\Psi\}\,.$$

That is, a logically atomic triple is equivalent to a normal triple where we do not directly get access to the precondition $P$, but instead to a resource $\mathsf{AU}_{P,Q}(\Psi)$ representing the right and obligation to update resources in $P$ into resources in $Q$ in one step. As witnessed by the universally quantified postcondition $\Psi$, when proving a logically atomic triple, one is indeed required to perform the atomic update (or, in other words, choose a linearization point) as it is the only way of obtaining $\Psi$.

Atomic updates are not themselves primitives to Iris, but for our purposes we will keep their definition abstract (more details can be found in [Jung et al. 2015b] and in the Coq formalization of Iris). Informally, it should be enough to remember that having ownership over an atomic update $\mathsf{AU}_{P,Q}$ gives us the liberty to choose a linearization point where we get access to resources $P$ and must update them into $Q$, while being able to access invariants.

### 6.2 Specification of a concurrent library

In this subsection we show what kind of logically atomic specification we consider for a library with a concurrent operation *op*. To simplify the presentation, we consider a library that just exposes a single operation, which can then be called concurrently by a client, possibly with different arguments. Note that this is in fact equivalent to considering a library exposing several operations: it is always possible for an operation to dispatch to several concurrent sub-operations depending on its argument.

Since we wish to cover a wide range of possible operations and specifications, we consider a fairly abstract specification, where a program value is related to a type $S$ of abstract states and where two mathematical functions $f$ and $r$ [4], of type $S \times \mathit{Val} \to S$ and $S \times \mathit{Val} \to \mathit{Val}$ respectively, are used to specify what the updated abstract state and return value should be after executing the operation, depending on its argument. Thus we will use a specification of the form:

$$\left\langle s.\, \mathit{PState}(x, s) \right\rangle\ \mathit{op}(x, y)\ \left\langle z.\, \mathit{PState}(x, f(s, y)) * \lceil z = r(s, y) \rceil \right\rangle.$$

Here $x$ is a pointer to the data structure that the library implements and which is accessed and modified by *op*; and $y$ is the argument passed to this specific call to *op*. *PState* corresponds to the abstract representation predicate describing the Separation Logic resources associated with $x$, and corresponding to the logical model $s$ of the data structure (for instance, for a concurrent stack data structure, $s$ would be the sequence of values that it stores). Notice that $s$ is bound in the precondition of the triple, instead of being bound on the outside as usual. This reflects the fact that the value of $s$ we are considering will only be known at the linearization point: because the execution of *op* might take several steps to execute, the value of $s$ might *change* during its execution, and thus cannot be predicted upfront.

Now, it is common for concurrent operations to rely on additional resources which are held in an invariant. To allow for that as well, we parameterize the specification further by a persistent

[4] $f$ and $r$ could be easily taken to be relations instead, a generalization that we have implemented in the Coq formalization.

predicate $IsP$, which we assume to hold. To relate the instances of $IsP$ and $PState$ that correspond to the same instantiation of the library, we use a ghost name $\gamma$. To sum up, $IsP(\gamma, x)$ can be shared between different threads, and expresses that the name $\gamma$ is associated with the structure at address $x$. $PState(\gamma, s)$ represents the knowledge that the current abstract state of the library is $s$, and is non-duplicable as it represents ephemeral information that gets updated by calling $op$. We thus refine the above specification to:

$$IsP(\gamma, x) \vdash \left\langle s.\ PState(\gamma, s) \right\rangle\ op(x, y)\ \left\langle z.\ PState(\gamma, f(s, y)) * \lceil z = r(s, y) \rceil \right\rangle.$$

We finally assume that we are provided an initial abstract state $s_{init} \in S$ for the library, and a function $op_{init}$ which, given initial resources $P_{init}$, initializes the library and returns a new instance.

Combining all these elements, our complete specification for a generic concurrent library is written as follows:

$$\begin{aligned}
&\Phi(op_{init}, op, (S, s_{init}, f, r), P_{init}, IsP, PState) \triangleq \\
&\quad \forall \gamma, x.\ \mathsf{persistent}(IsP(\gamma, x)) \wedge \\
&\quad \{P_{init}\}\ op_{init}()\ \{r.\ \exists \gamma.\ IsP(\gamma, r) * PState(\gamma, s_{init})\} \wedge \\
&\quad \forall \gamma, x, y.\ IsP(\gamma, x) \vdash \left\langle s.\ PState(\gamma, s) \right\rangle\ op(x, y)\ \left\langle z.\ PState(\gamma, f(s, y)) \wedge \lceil z = r(s, y) \rceil \right\rangle.
\end{aligned}$$

*Example: a concurrent counter library.* It can be useful to see a concrete example of the above generic specification. To this end, let us consider a simple counter library, exposing an init operation that creates a fresh counter, and an incr concurrent operation which increments the integer value held by the counter and returns its previous value. One can for instance implement incr by using the compare-and-swap primitive (CAS) to ensure that the increment occurs atomically. The counter library can be then specified as follows, using a logically atomic triple for incr:

$$\begin{aligned}
&\left\{\mathsf{True}\right\}\ \mathsf{init}()\ \left\{x.\ \exists \gamma, IsCnt(\gamma, x) * Cnt(\gamma, 0)\right\} \wedge \\
&\forall \gamma, x.\ IsCnt(\gamma, x) \vdash \left\langle n.\ Cnt(\gamma, n) \right\rangle\ \mathsf{incr}(x)\ \left\langle m.\ Cnt(\gamma, n+1) * \lceil m = n+1 \rceil \right\rangle,
\end{aligned}$$

where $Cnt$ and $IsCnt$ are abstract predicates and $IsCnt$ is persistent. By letting $\mathsf{incr}'(x, ()) \triangleq \mathsf{incr}(x)$, this counter specification can be seen as an instance of our generic specification above: it corresponds to $\Phi(\mathsf{init}, \mathsf{incr}', (\mathbb{N}, 0, succ_{\mathbb{N}}, succ_{Val}), \mathsf{True}, IsCnt, Cnt)$, where $succ_{\mathbb{N}}$ is the successor function on natural numbers, and $succ_{Val}$ is the successor function on program values representing integers.

### 6.3 Linearizability

Linearizability is the standard correcteness condition for (first-order) operations on thread-safe concurrent data structures [Herlihy and Wing 1990]. It expresses that, even if concurrent operations of a library are effectively interleaved at runtime, they behave the same as a sequence of sequential calls to the library. It can be formalized by requiring that between every call and return in a given interaction trace, there must exist a *single physical step* where the observable state of the data structure changes according to the specification of the operation. This point is known as the *linearization point*, and the behavior of the data structure must then match the sequential behavior of the operations when ordered according to the linearization points.

Let us define an alphabet $\Sigma$ of call and return events and an alphabet $\Sigma_{\mathsf{lin}}$ of call, linearization and return events as follows:

$$\begin{aligned}
\Sigma &\triangleq \{\langle \tau, \langle \mathtt{call}, v \rangle\rangle \mid \tau \in Tag,\ v \in Val\} \cup \{\langle \tau, \langle \mathtt{ret}, v, v' \rangle\rangle \mid \tau \in Tag,\ v, v' \in Val\} \\
\Sigma_{\mathsf{lin}} &\triangleq \Sigma \cup \{\langle \tau, \langle \mathtt{lin}, v, v' \rangle\rangle \mid \tau \in Tag,\ v, v' \in Val\}
\end{aligned}$$

Here $\tau$ is a tag used to relate calls, linearization and return events and $v$ and $v'$ are values representing the argument and return-value, respectively. We define the language $\mathcal{L}^{\tau}_{\mathsf{linfull}}$ of calls, linearizations and returns tagged with tag $\tau$ as follows. We then denote its prefix closure as $\mathcal{L}^{\tau}_{\mathsf{lin}}$.

$$\mathcal{L}^{\tau}_{\mathsf{linfull}} ::= \langle \tau, \langle \mathtt{call}, v\rangle\rangle \cdot \langle \tau, \langle \mathtt{lin}, v, v'\rangle\rangle \cdot \langle \tau, \langle \mathtt{ret}, v, v'\rangle\rangle \cdot \mathcal{L}^{\tau}_{\mathsf{linfull}} \mid \varepsilon$$

As before, we assume that the intended sequential behaviour of the library is given by a tuple $(S, s_{init}, (f, r))$ consisting of a set of abstract states $S$ along with two functions $(f, r)$ and an initial state $s_{init} \in S$. The function call $f(s, v)$ models a call to the operation in abstract state $s$ and with argument $v$; it returns the new abstract state. The function call $r(s, v)$ gives the return value of the operation when abstract state is $s$ and argument is $v$. A trace of linearization events satisfies the specification $(S, s_{init}, (f, r))$ if the return values in the trace match the intended return values given by $r$. This is captured by the sound predicate, defined below:

$$\mathsf{sound}(\alpha) \triangleq \exists s, \mathsf{soundWith}(\alpha, s_{init}, s)$$

where

$$\begin{array}{lcl} \mathsf{soundWith}(\varepsilon, s, s') & \triangleq & s = s', \\ \mathsf{soundWith}(\alpha \cdot \langle \mathtt{lin}, v, v_r\rangle, s, s'') & \triangleq & \exists s', \mathsf{soundWith}(\alpha, s, s') \wedge s'' = f(s', v) \wedge v_r = r(s', v) \end{array}$$

For a linearization trace $\beta \in \Sigma^*_{\mathsf{lin}}$, we write $\pi(\beta) \in \Sigma^*$ for the trace consisting of only the `call` and `ret` events of $\beta$; we write $\pi_\tau(\beta) \in \Sigma^*_{\mathsf{lin}}$ for the sub-trace of events with tag $\tau$; and $\pi_{\mathsf{lin}}(\beta) \in \mathit{Val}^*$ for the list of linearization events associated to `lin` events in $\beta$. We finally define the language $\mathcal{L}_{\mathsf{lin}}$ of linearizable traces as follows:

*Definition 6.1 (Linearizability).* An interaction trace $\alpha \in \Sigma^*$ is linearizable if there exists a trace $\beta \in \Sigma^*_{\mathsf{lin}}$ such that $\forall \tau.\, \pi_\tau(\beta) \in \mathcal{L}^{\tau}_{\mathsf{lin}}$, $\alpha = \pi(\beta)$, and $\mathsf{sound}(\pi_{\mathsf{lin}}(\beta))$.

### 6.4 Linearizability as a trace invariant

To prove that logical atomicity implies linearizability, we follow the usual procedure of introducing a wrapper that emits suitable trace events and proving that the wrapped library satisfies the same logically atomic specification as the original library and additionally a suitable invariant about the trace. The wrapping for the operation *op* is given by the following function:

$$\mathit{wrap}(\mathit{op}) \triangleq \lambda(x, y).\, \mathsf{let}\ \tau = \mathsf{fresh}\ \langle \mathtt{call}, y\rangle\ \mathsf{in}\ \mathsf{let}\ r = \mathit{op}(x, y)\ \mathsf{in}\ \mathsf{emit}\ \langle \tau, \langle \mathtt{ret}, y, r\rangle\rangle; r$$

This wrapping function emits suitable call and return events before and after the call to the underlying *op* operation. We use the fresh operation to generate a fresh identifier for every call event to allow us to uniquely associate return events with call events.

The following theorem then states the key correctness property: given any library satisfying a logically atomic specification, the wrapped library satisfies the same specification, and the trace it emits is linearizable.

Theorem 6.2.

$$\begin{array}{l} \forall \mathit{op}_{\mathit{init}}, \mathit{op}, \mathit{spec}, P_{\mathit{init}}, \mathit{IsP}, \mathit{PState}. \\ \quad \Phi(\mathit{op}_{\mathit{init}}, \mathit{op}, \mathit{spec}, P_{\mathit{init}}, \mathit{IsP}, \mathit{PState}) \Rightarrow \\ \quad \exists \mathit{IsP}_w, \mathit{PState}_w.\, \Phi(\mathit{op}_{\mathit{init}}, \mathit{wrap}(\mathit{op}), \mathit{spec}, P_{\mathit{init}} * \mathsf{trace}(\varepsilon) * \mathsf{traceInv}(\mathcal{L}_{\mathsf{lin}}), \mathit{IsP}_w, \mathit{PState}_w) \end{array}$$

As before, we can finally combine Theorem 6.2 with the Adequacy theorem (Theorem 4.1) and obtain as a free theorem that any library which admits a logically atomic specification produces an interaction trace in $\mathcal{L}_{\mathsf{lin}}$, i.e. is linearizable.

In the rest of this section, we detail the key ideas behind the proof of Theorem 6.2.

### 6.5 Proving linearizability

Our proof relies on two key ideas.

First, the library specification that we assume exposes that the operation *op* contains a linearization point during its execution. We can thus use this point in the execution as the moment where to record a `lin` event in the "ghost" linearization trace that we maintain as part of our trace invariant. Since the call to *op* in $wrap(op)$ occurs between the instructions emitting a `call` and `ret` event, we get that the `lin` event will indeed occur between the corresponding `call` and `ret` events.

Second, a key property we need to keep track of is that our use of tags and the fresh primitive indeed identify different calls to the wrapped operation with different tags. More precisely, we need to show that during the execution of $wrap(op)$, once a fresh tag has been generated, the currently running thread will be the only one emitting events with this tag. Thus, even though other threads can run concurrently, at every point of the execution of $wrap(op)$—after the call to fresh, after the call to *op*, and after the final emit—we know exactly what the linearization trace is *for events of that tag*: it is equal, respectively, to a single `call` event, a `call` event followed by a `lin` event, and finally a `call`, `lin`, and `ret` event.

We now show in more detail how these key ideas materialize in our proof.

*Tracking ghost state in Iris.* Our proof makes use of somewhat more advanced features of Iris, which we briefly introduce now. In particular, we define certain ghost theories (predicates and rules) on top of Iris' primitive notion of ghost state, which supports predicates of the form $\boxed{a}^{\gamma}$ that asserts the ownership of the resource $a$ at ghost name $\gamma \in Name$. Here the resource $a$ is an element of some resource algebra which restricts how it might be updated over time.

We use two constructions built on top of the Iris primitives for handling ghost state. How they are defined in terms of $\boxed{\cdot}^{\gamma}$ is not particularly relevant, as long as they satisfy a few rules, which will suffice for carrying out the rest of the proof and which we present below:

The first construction involves a pair of predicates $\mathsf{map}_\bullet$ and $\mathsf{map}_\circ$, used for tracking the state of a finite map $M$. Given a name $\gamma$ which identifies a specific instance of these predicates, ownership over $\mathsf{map}_\bullet(\gamma, M)$ asserts that the map of interest is exactly $M$, while $\mathsf{map}_\circ(\gamma, M')$ asserts that the map of interest is *at least* $M'$. The predicates satisfy the following rules, which we explain below.

GHOST-MAPFRAC-DUP
$$\mathsf{map}_\circ(\gamma, M) \vdash \mathsf{map}_\circ(\gamma, M) * \mathsf{map}_\circ(\gamma, M)$$

GHOST-MAP-SUB
$$\mathsf{map}_\bullet(\gamma, M) * \mathsf{map}_\circ(\gamma, M') \vdash \ulcorner M' \subseteq M \urcorner$$

GHOST-MAP-UPD
$$\mathsf{map}_\bullet(\gamma, M) * \mathsf{map}_\circ(\gamma, M) \vdash \mathsf{map}_\bullet(\gamma, M \uplus [a \mapsto b]) * \mathsf{map}_\circ(\gamma, M \uplus [a \mapsto b])$$

Given the combined knowledge of both predicates, one can deduce that $M'$ is a subset of $M$ (GHOST-MAP-SUB). In the case where both maps match, then we can update the map by monotonically extending it with a new key-value pair (GHOST-MAP-UPD). Finally, $\mathsf{map}_\circ(\gamma, M')$ is duplicable (GHOST-MAPFRAC-DUP), as it represents a "snapshot" of the state of the map at some point in the past, which remains sound as the map is extended monotonically.

The second construction involves a pair of predicates $\mathsf{val}_\bullet$ and $\mathsf{val}_\circ$, which are used to track the state of a discrete value of an arbitrary type from two different places. Ownership over either $\mathsf{val}_\bullet(\gamma, x)$ or $\mathsf{val}_\circ(\gamma, y)$ tells us that the value of interest is exactly $x$. The predicates satisfy the following rules, which we explain below.

GHOST-VAL-EQ
$$\mathsf{val}_\bullet(\gamma, x) * \mathsf{val}_\circ(\gamma, y) \vdash \ulcorner x = y \urcorner$$

GHOST-VAL-UPD
$$\mathsf{val}_\bullet(\gamma, x) * \mathsf{val}_\circ(\gamma, x) \vdash \mathsf{val}_\bullet(\gamma, z) * \mathsf{val}_\circ(\gamma, z)$$

When combining the knowledge of both predicates, we can deduce that the associated values match (GHOST-VAL-EQ) or update it to a new value (GHOST-VAL-UPD). Unlike $\mathsf{map}_\circ$, $\mathsf{val}_\circ$ is not duplicable as it needs to stay synchronized with the matching $\mathsf{val}_\bullet$ predicate.

*Wrapped representation predicates and main invariant.* Let us assume $op_{init}$, $op$, $(S, s_{init}, f, r)$, $IsP$ and $PState$ such that $\Phi(op_{\text{init}}, op, (S, s_{init}, f, r), P_{init}, IsP, PState)$ holds. The first step of the proof is then to pick suitable wrapped representation predicates $IsP_w$ and $PState_w$.

$$IsP_w \triangleq \lambda(\gamma, x).\, \exists \gamma_i, \gamma_s, \gamma_m.\, IsP(\gamma_i, x) * \mathsf{traceInv}(\mathcal{L}_{\mathsf{lin}}) * \boxed{\exists s.\, I(\gamma, \gamma_i, \gamma_s, \gamma_m, s)}$$
$$PState_w \triangleq \lambda(\gamma, s).\, \exists \gamma_i, \gamma_s, \gamma_m.\, PState(\gamma_i, s) * \mathsf{val}_\circ(\gamma, (\gamma_i, \gamma_s, \gamma_m)) * \mathsf{val}_\circ(\gamma_s, s)$$

Both $IsP_w$ and $PState_w$ take as argument a ghost name $\gamma$ which identifies a specific instance of the library. Because we need a few extra ghost names to track the various parts of ghost state that we need, we associate to $\gamma$ three ghost names $\gamma_i$, $\gamma_s$ and $\gamma_m$, using a $\mathsf{val}_\bullet$, $\mathsf{val}_\circ$ pair to ensure that they match between $IsP_w$ and $PState_w$. The $\mathsf{val}_\circ$ component of the pair appears in the definition of $PState_w$, and the $\mathsf{val}_\bullet$ component will appear next, in the definition of our key invariant $I$, which is itself held by $IsP_w$.

$IsP_w$, the persistent part of the representation predicate, holds the corresponding abstract $IsP$ predicate provided by the library specification, the $\mathsf{traceInv}(\mathcal{L}_{\mathsf{lin}})$ predicate enforcing that the trace is indeed linearizable at each step, and an additional Iris invariant $I$, which we define below. While $\mathsf{traceInv}(\mathcal{L}_{\mathsf{lin}})$ is ultimately the only property that we need to instantiate Theorem 4.1 and obtain that linearizability holds, during the proof we need to maintain additional invariants—held in $I$—to be able to show that it does hold as a trace invariant.

$PState_w$, which corresponds to knowledge that the library is in a specific state, asserts ownership over the underlying $PState$, and additionally holds halves of a $\mathsf{val}_\bullet$,$\mathsf{val}_\circ$ pair, associating respectively $\gamma$ to the auxiliary ghost names used, and associating the state of the library $s$ to ghost name $\gamma_s$. The corresponding $\mathsf{val}_\bullet$ halves appear next, in the definition of $I$.

$$\begin{aligned} I(\gamma, \gamma_i, \gamma_s, \gamma_m, s) \triangleq \exists \beta \in \Sigma^*_{\mathsf{lin}}.\, \exists M : Tag \rightharpoonup_{fin} Name.\ & \lceil \mathsf{dom}(M) = \mathsf{tags}(\pi(\beta)) \rceil * \\ & \mathsf{val}_\bullet(\gamma, (\gamma_i, \gamma_s, \gamma_m)) * \mathsf{val}_\bullet(\gamma_s, s) * \mathsf{map}_\bullet(\gamma_m, M) * \mathsf{map}_\circ(\gamma_m, M) * \\ & \mathsf{trace}(\pi(\beta)) * \lceil \mathsf{sound}(\pi_{\mathsf{lin}}(\beta)) \wedge \forall \tau.\, \pi_\tau(\beta) \in \mathcal{L}^\tau_{\mathsf{lin}} \rceil * \\ & \mathsf{tagsState}(M, \beta) \end{aligned}$$

The trace $\beta$ is the candidate linearization trace, composed of `call`, `lin` and `ret` events, such that its restriction to `call` and `ret` events $\pi(\beta)$ corresponds to the current physical trace.

The finite map $M$ associates a ghost name for each tag currently appearing in the trace. As mentioned before, for each fresh tag that has been allocated, we wish to track the point of the execution where the corresponding wrapper is. $M$ gets us halfway there: for each tag, it associates a ghost name which can then be used to track precisely the state of the wrapper as a ghost resource. This indirection allows $M$ to grow monotonically as new tags are emitted, while updates to the state of a wrapper can be handled separately by updating the corresponding ghost resource.

The invariant $I$ holds a half of a $\mathsf{val}_\bullet$,$\mathsf{val}_\circ$ pair for both the auxiliary ghost names and the state of the library $s$, matching the other halves held by the $PState_w$ predicate. It also holds both $\mathsf{map}_\bullet$ and $\mathsf{map}_\circ$ resources for tracking the state of the map $M$. This serves two purposes: first, it allows updating the map when a new tag is emitted by fresh using rule GHOST-MAP-UPD; second, it allows handing out afterwards copies of $\mathsf{map}_\circ(\gamma_m, M \uplus [\tau \mapsto \gamma_\tau])$ (using rule GHOST-MAPFRAC-DUP) to serve as witnesses that a tag $\tau$ has been emitted and is associated to a ghost name $\gamma_\tau$.

Invariant $I$ holds the resource $\mathsf{trace}(\pi(\beta))$ granting the exclusive permission to update the physical trace. Reasoning about the calls to the fresh and emit primitive—which update this

resource—will thus require opening $I$ and re-establish it afterwards. $I$ moreover enforces that the linearization events are sound, and that $\beta$ is per-tag linearized, as required to prove linearizability.

Finally, the assertion $\mathsf{tagsState}(M, \beta)$ (which we define now) describes the collection of ghost resources tracking the indiviual state of wrappers for each tag in $M$, relating it to the current linearization trace $\beta$.

$$
\begin{array}{l}
\mathit{TagState} \triangleq \mathrm{AfterCall}(v) \mid \mathrm{AfterLin}(v, v') \mid \mathrm{Done} \\
\mathsf{tagsState}(M, \beta) \triangleq \\
\quad \displaystyle\mathop{\Large *}_{(\tau,\gamma_\tau)\in M} \left( \begin{array}{l} \exists \zeta : \mathit{TagState}.\ \mathsf{val}_\bullet(\gamma_\tau, \zeta) * \\ \quad \left[ \begin{array}{ll} \text{match } \zeta \text{ with} & \\ \mid \mathrm{AfterCall}(v) & \Rightarrow \pi_\tau(\beta) = \langle \tau, \langle \mathtt{call}, v \rangle \rangle \\ \mid \mathrm{AfterLin}(v, v') & \Rightarrow \pi_\tau(\beta) = \langle \tau, \langle \mathtt{call}, v \rangle \rangle \cdot \langle \tau, \langle \mathtt{lin}, v, v' \rangle \rangle \\ \mid \mathrm{Done} & \Rightarrow \mathrm{True} \end{array} \right] \end{array} \right)
\end{array}
$$

Values of type *TagState*, here defined as an inductive datatype, describe the state of a thread running an instance of $\mathit{wrap}(\mathit{op})$: $\mathrm{AfterCall}(v)$ indicates that its execution is after a call to $\mathsf{fresh}\langle \mathtt{call}, v \rangle$ but before the linearization point; $\mathrm{AfterLin}(v, v')$ indicates that it is after a linearization point $\langle \mathtt{lin}, v, v' \rangle$ but before the final emit; and Done indicates that it is done, i.e. after the final emit. Then, for each pair of a tag $\tau$ and a ghost name $\gamma_\tau$ in $M$, $\mathsf{tagsState}(M, \beta)$ names $\zeta$ the state *TagState* of the thread which has first allocated $\tau$, and firstly, ties it with the ghost name $\gamma_\tau$ using the assertion $\mathsf{val}_\bullet(\gamma_\tau, \zeta)$; then secondly, enforces that the value of $\zeta$ indeed gives us the expected information about events emitted so far with tag $\tau$ in $\beta$.

The matching assertion $\mathsf{val}_\circ(\gamma_\tau, \zeta)$ is not stored inside the invariant $I$. Instead, it will be kept "on the side" in the proof context, as a token expressing that the current state for tag $\tau$ is exactly $\zeta$, for a concrete value of $\zeta$. This token grants the unique, exclusive permission to emit new events with tag $\tau$, updating the state $\zeta$ in the process. Consequently, owning such a token during the proof of $\mathit{wrap}(\mathit{op})$ guarantees that no other thread is able to emit events with the current tag of interest: this is one of the two key properties that we need to make the proof work.

*Outline of the proof.* There are three main steps to the proof: one to reason about the call to fresh, one for calling to the underlying operation *op*, and one for reasoning about the final emit.

At the beginning of the proof, we are given $\mathit{IsP}_w(\gamma, x)$, and are asked to prove a logically atomic triple. We first use the rule LOGATOM-AU to turn this triple into a normal Hoare triple:

$$
\begin{array}{l}
\left\{ \mathit{IsP}_w(\gamma, x) * \mathsf{AU}_{(\forall s.\mathit{PState}_w(\gamma,s)),(\mathit{PState}_w(\gamma,f(s,y)))}(\Psi) \right\} \\
\quad \mathsf{let}\ \tau = \mathsf{fresh}\ \langle \mathtt{call}, y \rangle\ \mathsf{in}\ \mathsf{let}\ r = \mathit{op}(x, y)\ \mathsf{in}\ \mathsf{emit}\ \langle \tau, \langle \mathtt{ret}, y, r \rangle \rangle; r \\
\{\Psi\}
\end{array}
$$

We can then start and step through the wrapper code:

- Call to $\mathsf{fresh}\ \langle \mathtt{call}, y \rangle$.
  Because fresh is physically atomic (as a primitive of the language), this reasoning step follows the usual pattern of reasoning with invariants in concurrent separation logic: we unfold $\mathit{IsP}_w$ and open the invariant containing $I$ around the instruction. We thus get access to $I(\gamma, \gamma_i, \gamma_s, \gamma_m, s)$ for some $\gamma_i, \gamma_s, \gamma_s$ and $s$, which in turn gives us access to the trace resource required to call fresh. It is easy to check that adding a `call` event with a fresh tag preserves the linearizability of the trace.
  Afterwards we need to close the invariant. Because we added a new tag to the trace, we need to extend the tagsState resource it contains to account for the new tag. We thus allocate a new pair $\mathsf{val}_\bullet(\gamma_\tau, \mathrm{AfterCall}(y)), \mathsf{val}_\circ(\gamma_\tau, \mathrm{AfterCall}(y))$ for a fresh $\gamma_\tau$.

After closing the invariant, we obtain the following extra resources as witnesses of the operation, to keep around in our proof context for later:

$$\exists M, \gamma_\tau.\, \mathsf{map}_\circ(\gamma_m, M) * \lceil M(\tau) = \gamma_\tau \rceil * \mathsf{val}_\circ(\gamma_\tau, \mathsf{AfterCall}(y))$$

- Call to $op(x, y)$.
  Since $op$ is logically atomic, we can again open the invariant containing $I$. Then, we need to establish the atomic update from the specification of $op$ ($\mathsf{AU}_{(\forall s.PState(\gamma_i,s)),(PState(\gamma_i,f(s,y)))}$) using the one we have in the context ($\mathsf{AU}_{(\forall s.PState_w(\gamma,s)),(PState_w(\gamma,f(s,y)))}$).
  This is the linearization point. Given a way of updating in one step the state of the unwrapped operation $PState(\gamma_i, s)$, we need to show that we can similarly update the state of the wrapped operation $PState_w(\gamma, s)$, while preserving our invariants.
  Updating $PState_w(\gamma, s)$ to $PState_w(\gamma, f(s, y))$ requires updating the $\mathsf{val}_\circ(\gamma_s, s)$ resource it contains. This is easy: we have the corresponding $\mathsf{val}_\bullet$ resource available as part of the opened invariant.
  Then, thanks to the $\mathsf{val}_\circ(\gamma_\tau, \mathsf{AfterCall}(y))$ resource that we have in the context, we can deduce that the current trace only contains one `call` event with tag $\tau$. We can then show that $I(\gamma, \gamma_i, \gamma_s, \gamma_m, s)$ can be reformulated into $I(\gamma, \gamma_i, \gamma_s, \gamma_m, f(s, y))$ by recording a $\langle \mathtt{lin}, y, r(s, y)\rangle$ event in the linearization trace, and that linearizability is preserved.
  In the process, the state associated with $\gamma_\tau$ has to be updated, which means that we now hold $\mathsf{val}_\circ(\gamma_\tau, \mathsf{AfterLin}(y, r(s, y)))$ in the context.

- Call to emit $\langle \tau, \langle \mathtt{ret}, y, r\rangle\rangle$.
  As with the call to fresh, this is a call to a physically atomic operation, so we can reason as usual by opening then closing the main invariant. Because we hold $\mathsf{val}_\circ(\gamma_\tau, \mathsf{AfterLin}(y, r(s, y)))$, we deduce that the current linearization trace contains a `call` event followed by a `lin` event. This is enough to show that emitting a corresponding `ret` event preserves both the invariant $I$ and the linearizability of the trace, thus completing the proof.

### 6.6 Related Work

Our key contribution is a technique for deriving free theorems from Separation Logic specifications, by formally relating separation logic specifications with the temporal properties they enforce, through wrapping of abstract resources with assertions about traces. As alluded to in the introduction, the idea of deriving free theorems from specifications is akin to the idea of deriving free theorems from polymorphic types [Reynolds and Plotkin 1993; Reynolds 1983; Wadler 1989]. In both cases, the term "free theorem" is used to highlight that a theorem is obtained from the specification / type; in particular, it holds for all programs satisfying the specification / type. In both cases, a "free theorem" needs proof. In the case of polymorphic types, one often uses a relationally parametric model of types (often formulated using logical relations) and the key point in the proof is then to pick a suitable relation for the quantified type variable. In our case, the key point in the proof is to wrap the assumed library implementation with operations that generate a suitable trace of the interactions between a library and a client and then show that the wrapped library satisfies the specification.

We now discuss some further related work not already discussed earlier in the paper.

*Approaches based on type systems.* Static analyses based on type systems have been widely used to check resource usage, like our file module example, and more generally enforce interaction protocols.

A particularly influential approach has been Typestates [DeLine and Fähndrich 2004; Strom and Yemini 1986]. These can be seen as specifying trace properties using pre/post-conditions. They have

been used to check safety temporal properties of programs, by associating abstract states to objects, then specifying which methods can be called at each state and how they make the state evolve. In [Bierhoff and Aldrich 2007; Bierhoff et al. 2009] they are combined with aliasing information to give a sound and modular analysis of API usage protocols, and to check a specification for the iterator module similar to ours, yet based on a single object. Multi-object properties, like the iterator one, can be captured by an extension of typestates using tracematches to specify intensional properties [Naeem and Lhoták 2008].

Session types [Honda et al. 1998; Vasconcelos 2012] are another way of enforcing interaction patterns between interacting program components. Contracts have also been studied in the context of session types to check for safety trace-like properties [Gommerstadt et al. 2018]. Properties of interest are then checked by monitors—wrappers that run alongside the code—reminiscent of our wrappers emitting events. The main difference being then that monitors run dynamic checks (without changing the results of the instrumented program), while our wrappers only expose the interactions between library and context by emitting events, the trace properties of interest being proved statically.

Linear types are used in [DeLine and Fähndrich 2001] to develop Vault, a programming language used to design device drivers, where resource management protocols can be specified explicitly using annotations in the source code. An automatic analysis has then been developed in [Igarashi and Kobayashi 2002]. One a more foundational level, core calculus based on linear types have been studied to support language features related to resource management, such as strong updates [Morrisett et al. 2005] or the mechanisms for safe memory management in low-level systems code used in the Cyclone language [Fluet et al. 2006].

System F° [Mazurak et al. 2010], an extension of System F with linear types, is shown to be expressive enough to encode as a F° type any protocol representable as a DFA. The authors show a parametricity result for the language, and show in particular the example of a file interface similar to ours. The operational semantics of the language also uses traces to track the flow of values through the program (when they are passed to a function, and when they are used). The authors show that their system can encode typestate-like properties, by proving (on paper) that any context using a value of a type encoding a protocol (as a DFA) will indeed always use it in a way that satisfies the corresponding DFA. They however do not exhibit a systematic proof method for establishing results similar to our "free theorems", and the expressivity of their type system is limited compared to the expressive separation logic that we consider in this work.

Type and effect systems have been used in [Skalka et al. 2008] to infer resource usage, represented by an LTS, and combined with model checking to verify trace properties of programs. Such systems have been applied to Featherweight Java in [Skalka 2008], where challenges coming from object orientation like inheritance and dynamic dispatch are tackled.

Higher-order model checking has been used to provide a sound and complete resource usage analysis, using higher-order recursion scheme model checking for a fragment of the $\mu$-calculus [Kobayashi 2009].

The Mezzo programming language [Balabonski et al. 2016; Pottier and Protzenko 2015] uses a type system based on ideas from Separation Logic, and as such is able to express library specifications in a style similar to ours. The work on Mezzo however focuses on the design and expressivity of its type system, and does not study which "free theorems" could be deduced from its types.

Such approaches based on type systems have been typically designed to support automated static verification and thus trade off expressiveness for automation. In this work we have made the opposite trade-off and focused on being able to capture expressive trace properties. As seen, we can specify non-regular properties (§5.3, the language is visibly pushdown [Alur and Madhusudan 2004]), others that rely on tracking an unbounded number of objects (§5.2, where we need to track

all valid iterators), and we can even go beyond context-free languages (§5.4, the language requires an order-2 pushdown automaton [Maslov 1976]). In an orthogonal direction, working with a logic with quantification we can specify traces from infinite alphabets of trace events (§5.2, §5.3, and §5.4).

*Approaches based on program logics.* We have mentionned in the introduction a number of approaches relying on program logics to establish temporal trace properties. These approaches typically verify trace-oriented specifications for library implementations; in contrast, our work establishes temporal trace properties from separation logic *specifications* that are not inherently temporal.

We additionally mention here the F7 and F* programming languages, which provide another technique for reasoning about trace properties [Bengtson et al. 2011; Swamy et al. 2013]. The technique is primarily aimed at verifying cryptographic primitives, but has also been applied to access control policies about interactions between a client and resources managed through libraries [Borgström et al. 2011]. It is based on extending the base programming language with a primitive for assuming that a given formula holds and an assert primitive that fails if a given formula does not follow from all previously assumed formulas. Access control policies are encoded by inserting appropriate assume and assert statements and proving that no assert can fail. In contrast to our work, the F7 and F* approach does not establish a formal connection between the inserted assume/assert statements and the property enforced on the execution. It is also non-local in that any assume statement can introduce a contradiction and break adequacy.

Finally, we mention the more recent work on using interaction trees [Xia et al. 2019] in combination with the VST framework [Cao et al. 2018] to verify the trace behavior of a networked server [Koh et al. 2019]. In that work, interaction trees are used to specify traces of behaviors for C programs, verified using Separation Logic, where the trace of a given program records external calls made by that program. Similarly to our traceInv predicate (parameterized by a set of traces), the authors rely on an ITree Separation Logic predicate (parameterized by an interaction tree) to specify the trace behavior of a given program. The authors also establish a form of linearizability property, as a refinement between interaction trees specifying the network behavior of the web server. The concrete interaction tree specifying a network behavior with possibly interleaved events is shown to refine a simpler interaction tree where network events have been linearized. However, since this connection is made purely at the level of interaction trees, it is only superficially related to our result of Section 6, where we relate linearizability as a trace property and logically atomic triples which do not involve traces.

## 7 CONCLUSION

We have presented a formal approach for deriving free theorems from Separation Logic specifications. We have focused on free thorems expressing trace properties on the interaction between clients verified against the specified library and the library itself. Since our main goal has been to establish a theoretical foundation relating specifications and trace properties, we focused on expressiveness rather than automation.

## REFERENCES


Rajeev Alur and P. Madhusudan. 2004. Visibly pushdown languages. In *Proceedings of the 36th Annual ACM Symposium on Theory of Computing, Chicago, IL, USA, June 13-16, 2004*. 202–211.

Thibaut Balabonski, François Pottier, and Jonathan Protzenko. 2016. The Design and Formalization of Mezzo, a Permission-Based Programming Language. *ACM Trans. Program. Lang. Syst.* 38, 4, Article 14 (Aug. 2016), 94 pages. https://doi.org/10.1145/2837022

Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. 2011. Refinement Types for Secure Implementations. *ACM Trans. Program. Lang. Syst.* 33, 2 (Feb. 2011), 8:1–8:45.

Kevin Bierhoff and Jonathan Aldrich. 2007. Modular Typestate Checking of Aliased Objects. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications (OOPSLA'07)*. ACM, 301–320.

Kevin Bierhoff, Nels E Beckman, and Jonathan Aldrich. 2009. Practical API Protocol Checking with Access Permissions. In *Proceedings of the 23rd European Conference on Object-Oriented Programming (ECOOP'09)*. Springer-Verlag, 195–219.

Bodil Biering, Lars Birkedal, and Noah Torp-Smith. 2007. BI-hyperdoctrines, Higher-order Separation Logic, and Abstraction. *ACM Trans. Program. Lang. Syst.* 29, 5, Article 24 (Aug. 2007).

Lars Birkedal and Aleš Bizjak. 2020. Lecture Notes on Iris: Higher-Order Concurrent Separation Log. (2020). http://iris-project.org/tutorial-pdfs/iris-lecture-notes.pdf

Johannes Borgström, Andrew D. Gordon, and Riccardo Pucella. 2011. Roles, Stacks, Histories: A Triple for Hoare. *Journal of Functional Programming* 21 (2011), 159–207. Issue 02.

Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W. Appel. 2018. VST-Floyd: A Separation Logic Tool to Verify Correctness of C Programs. *Journal of Automated Reasoning* 61, 1-4 (2018), 367–422. https://doi.org/10.1007/s10817-018-9457-5

Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. 2014. TaDA: A Logic for Time and Data Abstraction. In *Proceedings of ECOOP*.

Robert DeLine and Manuel Fähndrich. 2001. Enforcing High-level Protocols in Low-level Software. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI '01)*. ACM, 59–69.

Robert DeLine and Manuel Fähndrich. 2004. Typestates for Objects. In *Object-Oriented Programming: Proceedings of the 18th European Conference on Object-Oriented Programming (ECOOP'04)*. Springer-Verlag, 465.

Matthew Fluet, Greg Morrisett, and Amal Ahmed. 2006. Linear Regions Are All You Need. In *Programming Languages and Systems*, Peter Sestoft (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 7–21.

Ming Fu, Yong Li, Xinyu Feng, Zhong Shao, and Yu Zhang. 2010. Reasoning about Optimistic Concurrency Using a Program Logic for History. In *In Proceedings of CONCUR*.

Philippa Gardner, Gian Ntzik, and Adam Wright. 2014. Local Reasoning for the POSIX File System. In *ESOP*. 169–188.

Hannah Gommerstadt, Limin Jia, and Frank Pfenning. 2018. Session-Typed Concurrent Contracts. In *Programming Languages and Systems*, Amal Ahmed (Ed.). Springer International Publishing, Cham, 771–798.

Alexey Gotsman, Noam Rinetzky, and Hongseok Yang. 2013. Verifying Concurrent Memory Reclamation Algorithms with Grace. In *Proceedings of ESOP*.

Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (July 1990), 463–492.

Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. 1998. Language primitives and type discipline for structured communication-based programming. In *Programming Languages and Systems*, Chris Hankin (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 122–138.

Atsushi Igarashi and Naoki Kobayashi. 2002. Resource Usage Analysis. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '02)*. ACM, 331–342.

Samin S. Ishtiaq and Peter W. O'Hearn. 2001. BI As an Assertion Language for Mutable Data Structures. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'01)*. ACM, 14–26.

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015a. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. 637–650. https://doi.org/10.1145/2676726.2676980

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015b. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. 637–650. https://doi.org/10.1145/2676726.2676980

Naoki Kobayashi. 2009. Types and Higher-order Recursion Schemes for Verification of Higher-order Programs. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '09)*. ACM, 416–428.

Nicolas Koh, Yao Li, Yishuai Li, Li-yao Xia, Lennart Beringer, Wolf Honoré, William Mansky, Benjamin C. Pierce, and Steve Zdancewic. 2019. From C to Interaction Trees: Specifying, Verifying, and Testing a Networked Server. In *Proceedings of the*

*8th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Cascais, Portugal) *(CPP 2019)*. Association for Computing Machinery, New York, NY, USA, 234–248. https://doi.org/10.1145/3293880.3294106

Neelakantan R. Krishnaswami, Jonathan Aldrich, Lars Birkedal, Kasper Svendsen, and Alexandre Buisse. 2009. Design Patterns in Separation Logic. In *Proceedings of the 4th International Workshop on Types in Language Design and Implementation (TLDI'09)*. ACM, 105–116.

A. N. Maslov. 1976. Multilevel stack automata. *Problems Inform. Transmission* 12, 1 (1976), 38–42.

Karl Mazurak, Jianzhou Zhao, and Steve Zdancewic. 2010. Lightweight Linear Types in System F°. In *Proceedings of the 5th ACM SIGPLAN Workshop on Types in Language Design and Implementation* (Madrid, Spain) *(TLDI '10)*. Association for Computing Machinery, New York, NY, USA, 77–88. https://doi.org/10.1145/1708016.1708027

Greg Morrisett, Amal Ahmed, and Matthew Fluet. 2005. L3: A Linear Language with Locations. In *Typed Lambda Calculi and Applications*, Paweł Urzyczyn (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 293–307.

Nomair A Naeem and Ondrej Lhoták. 2008. Extending typestate analysis to multiple interacting objects. *OOPSLA'08: Proceedings of Object-Oriented Programming, Systems, Languages and Applications* (2008).

Matthew Parkinson and Gavin Bierman. 2005. Separation Logic and Abstraction. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'05)*. ACM, 247–258.

François Pottier and Jonathan Protzenko. 2015. A few lessons from the Mezzo project. In *Summit oN Advances in Programming Languages (SNAPL) (Leibniz International Proceedings in Informatics, Vol. 32)*. Asilomar, United States. https://doi.org/10.4230/LIPIcs.SNAPL.2015.221

J.C. Reynolds and G.D. Plotkin. 1993. On Functors Expressible in the Polymorphic Typed Lambda Calculus. *Information and Computation* 105, 1 (1993), 1–29. https://www.sciencedirect.com/science/article/pii/S0890540183710370

John C. Reynolds. 1983. Types, Abstraction and Parametric Polymorphism. In *Information Processing 83*. 513–523. http://www.cse.chalmers.se/edu/year/2010/course/DAT140_Types/Reynolds_typesabpara.pdf

John C Reynolds. 2002. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS'02)*. IEEE, 55–74.

Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. 2015. Specifying and Verifying Concurrent Algorithms with Histories and Subjectivity. In *Proceedings of ESOP*.

Christian Skalka. 2008. Types and trace effects for object orientation. *Higher-Order and Symbolic Computation* 21, 3 (2008), 239–282.

Christian Skalka, Scott Smith, and David Van Horn. 2008. Types and trace effects of higher order programs. *Journal of Functional Programming* 18, 02 (2008), 179–249.

Robert E Strom and Shaula Yemini. 1986. Typestate: A Programming Language Concept for Enhancing Software Reliability. *IEEE Trans. Softw. Eng.* 12, 1 (Jan. 1986), 157–171.

Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. 2013. Secure Distributed Programming with Value-Dependent Types. *Journal of Functional Programming* 23, 4 (2013), 402–451.

Vasco T. Vasconcelos. 2012. Fundamentals of session types. *Information and Computation* 217 (2012), 52–70. https://doi.org/10.1016/j.ic.2012.05.002

Philip Wadler. 1989. Theorems for free! 347–359. http://homepages.inf.ed.ac.uk/wadler/papers/free/free.ps.gz

Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. 2019. Interaction Trees: Representing Recursive and Impure Programs in Coq. *Proc. ACM Program. Lang.* 4, POPL, Article 51 (Dec. 2019), 32 pages. https://doi.org/10.1145/3371119