Trace Properties from Separation Logic Specifications

Birkedal, L; Dinsdale-Young, T; Jaber, G; Svendsen, K; Tzevelekos, N

https://arxiv.org/abs/1702.02972#

Trace Properties from Separation Logic Specifications

Lars Birkedal[1], Thomas Dinsdale-Young[1], Guilhem Jaber[2], Kasper Svendsen[3], and Nikos Tzevelekos[4]

[1] Aarhus University ({birkedal,tyoung}@cs.au.dk)
[2] University Paris Diderot (gjaber@pps.univ-paris-diderot.fr)
[3] University of Cambridge (ksvendsen@cs.au.dk)
[4] Queen Mary University of London (nikos.tzevelekos@qmul.ac.uk)

Abstract. We propose a formal approach for relating abstract separation logic library specifications with the trace properties they enforce on interactions between a client and a library. Separation logic with abstract predicates enforces a resource discipline that constrains when and how calls may be made between a client and a library. Intuitively, this can enforce a protocol on the interaction trace. This intuition is broadly used in the separation logic community but has not previously been formalised. We provide just such a formalisation. Our approach is based on using wrappers which instrument library code to induce execution traces for the properties under examination. By considering a separation logic extended with trace resources, we prove that when a library satisfies its separation logic specification then its wrapped version satisfies the same specification and, moreover, maintains the trace properties as an invariant. Consequently, any client and library implementation that are correct with respect to the separation logic specification will satisfy the trace properties.

1 Introduction

Separation logic [24,15] 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. Abstract predicates [23] crucially hide the implementation details — the client is not aware of what an "opened file" resource actually consists of, which may vary between implementations of the library, but only the functionality for using it 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 be the following:

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

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:

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

On the other hand, we cannot prove any useful specifications[5] 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. If the client were able to forge the `open` resource then it could violate the trace property. 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 trace properties from abstract separation logic specifications. We achieve this by placing a *wrapper* between a client and a library that generates a trace of the interactions between the client and library. The wrapper has no bearing on the underlying semantics of the program (when traces are ignored) but simply allows us to formally interpret trace properties. Supposing that the library implements an abstract separation logic specification, we show that the wrapped library also satisfies this specification, but moreover maintains the desired trace properties as an invariant. For the latter step, we recur to a separation logic extended with 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 (q.v. §7), it is the one to derive trace properties for libraries that already have separation logic specifications. Previous approaches [11,9,26,25] achieve temporal reasoning by specifying and verifying the underlying libraries using trace-oriented specifications; the novelty of our work is in deriving temporal properties from trace-agnostic specifications.

A motivation for our approach is to *justify* separation logic specifications, by showing that they entail more intuitive trace properties. Such justification is particularly useful if the specification is intended to formally capture an English-language specification (e.g. the POSIX file system [10]). While we limit our presentation to the sequential setting and a simple higher-order separation logic, we

[5] It is always possible to prove a vacuous specification with precondition $\perp$, but a useful specification should at least have a satisfiable precondition. For complete, closed programs we would typically expect the precondition to be emp.

can show trace properties that include elaborate protocols that are, for instance, parametric on object identifiers or beyond regular languages (q.v §5).

We begin by considering an illustrative example in §2. We introduce the programming language (§3) and our separation logic (§4) before we tackle examples in depth in §5. These examples establish that separation logic specifications can imply a variety of trace properties, 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).
– Any string received from the user should be sanitised before it is included in an SQL query (§A).

The semantics of the logic is presented in §6. Finally, we discuss conclusions and related work in §7.

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:

$$\forall \alpha, a.\ \{\mathsf{stack}(\alpha) \wedge a \neq ()\}\ \mathtt{push}(a)\ \{\mathsf{stack}(a :: \alpha)\}$$
$$\forall \alpha.\ \left\{\mathsf{stack}(\alpha)\right\}\ \mathtt{pop}()\ \left\{\begin{matrix} r.\,(r = () \wedge \mathsf{stack}(\alpha) \wedge \alpha = \varepsilon) \vee \\ (\exists \alpha'.\, \alpha = r :: \alpha' \wedge \mathsf{stack}(\alpha')) \end{matrix}\right\}$$

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 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.

In order capture trace properties, we define a wrapper that instruments the library operations with code to emit trace events as:

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

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

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

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 $\mathsf{inv}(I)$ 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:

$$\mathsf{stack}'(\alpha) \stackrel{\text{def}}{=} \mathsf{stack}(\alpha) * \mathsf{inv}(\mathcal{L}) * \exists t.\, \mathsf{trace}(t) \wedge t \in \mathcal{L} \wedge \forall a \in \alpha.\, \exists i.\, t[i] = \langle \mathtt{push}, a\rangle$$

The proof of the wrapped push operation proceeds as follows:

$\{\mathsf{stack}'(\alpha) \wedge v \neq ()\}$
$\{\mathsf{stack}(\alpha) \wedge v \neq () * \mathsf{inv}(\mathcal{L}) * \exists t.\, \mathsf{trace}(t) \wedge t \in \mathcal{L} \wedge \forall a \in \alpha.\, \exists i.\, t[i] = \langle \mathtt{push}, a\rangle\}$
  push(v);
$\{\mathsf{stack}(v :: \alpha) * \mathsf{inv}(\mathcal{L}) * \exists t.\, \mathsf{trace}(t) \wedge t \in \mathcal{L} \wedge \forall a \in \alpha.\, \exists i.\, t[i] = \langle \mathtt{push}, a\rangle\}$
$\{\mathsf{stack}(v :: \alpha) * \mathsf{inv}(\mathcal{L}) * \exists t.\, \mathsf{trace}(t) \wedge (t \cdot \langle \mathtt{push}, v\rangle) \in \mathcal{L} \wedge$
  $\forall a \in (v :: \alpha).\, \exists i.\, (t \cdot \langle \mathtt{push}, v\rangle)[i] = \langle \mathtt{push}, a\rangle\}$
  emit⟨push,v⟩;
$\{\mathsf{stack}(v :: \alpha) * \mathsf{inv}(\mathcal{L}) * \exists t.\, \mathsf{trace}(t) \wedge t \in \mathcal{L} \wedge \forall a \in \alpha.\, \exists i.\, t[i] = \langle \mathtt{push}, a\rangle\}$
$\{\mathsf{stack}'(v :: \alpha)\}$

The wrapped pop operation can be verified similarly. We can thus conclude that the stack indeed satisfies the desired trace property.

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:

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

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 $I(rev(\alpha))$, where $rev(\alpha)$ is the reversal of the list α. 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 $I(rev(\alpha))$ predicate for the postcondition from the $I(\varepsilon)$ given in the precondition, it must call f on each element of α 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 define the programming language which will be the object of this study. To keep the setting general, the language is an untyped call-by-value λ-calculus with references, named functions, pairs, integers and primitives for arithmetic, reference usage and conditionals. Our language additionally features an emit primitive, which is used to output values as trace events. We will formulate properties in terms of the event traces produced by program evaluation. The syntactic classes of *values* (*Val*), *expressions* (*Exp*) and *evaluation contexts* (*Con*) are given respectively as follows.

$$\begin{aligned}
u, v &::= () \mid n \mid x \mid l \mid f \mid \langle u, v\rangle \\
e &::= v \mid \lambda x.e \mid e\ e' \mid \mathsf{if}\, e_1\, e_2\, e_3 \mid \langle e, e'\rangle \mid \pi_i(e) \mid \mathsf{ref}\, e \mid !e \mid e\,\mathsf{op}\, e' \mid \mathsf{emit}\, v \\
K &::= \bullet \mid K e \mid v K \mid K\,\mathsf{op}\, e \mid v\,\mathsf{op}\, K \mid \mathsf{if}\, K\, e\, e' \mid \langle K, e\rangle \mid \langle v, K\rangle \mid \pi_i(K) \mid \mathsf{ref}\, K \mid !K
\end{aligned}$$

Above we let $i \in \{1, 2\}, n \in \mathbb{Z}, f \in \mathit{Fun}$ and $l \in \mathit{Loc}$, where *Fun* and *Loc* are countable sets of function and location identifiers respectively. Moreover, x ranges over the set of variables *Var*, and $\mathsf{op} \in \{=, :=, \dots\}$ ranges over a set of binary operators which includes equality test, assignment and arithmetic operators. We use *Lam* for the set of λ-abstraction expressions. Note that λ-abstractions are not values; functional values are represented by function identifiers.

Our operational semantics tracks λ-abstractions in an environment $\gamma : \mathit{Fun} \rightharpoonup_{\mathrm{fin}} \mathit{Lam}$, where all encountered functions are named and stored. This allows us to track function usage by emitting trace events that refer to these function names. In addition, we can reduce expressions that use external functions: in such a case, an expression e can contain the names of the external functions in its code, and

γ would provide their bodies. The semantics draws from the open-term trace semantics used e.g. in [20], the main difference being that we explicitly control events generated by the reduction via the emit primitive (and also indiscriminately name λ-abstractions).

Expressions are evaluated inside states $(h, \gamma) \in \mathit{Heap} \times \mathit{FEnv}$ comprising a function environment γ and a heap h. A heap is a finite map from locations to values and a function environment is a finite map from function identifiers to λ-abstractions. For any domain/codomain set pair X, Y, map $g : X \rightharpoonup Y$ and $(x, y) \in X \times Y$, we let $g[x \mapsto y] = \{(x, y)\} \cup \{\, (z, g(z)) \mid z \in \mathsf{dom}(g) \setminus \{x\} \,\}$ regardless of whether $x \in \mathsf{dom}(g)$ or not. The evaluation rules are:

$$\begin{array}{rcll}
K[\lambda x.e], (h,\gamma) & \to & K[f], (h, \gamma[f \mapsto \lambda x.\, e]) & (f \notin \mathsf{dom}(\gamma)) \\
K[f\, v], (h,\gamma) & \to & K[e\{v/x\}], (h,\gamma) & (\gamma(f) = \lambda x.\, e) \\
K[\pi_i(\langle v_1, v_2\rangle)], (h,\gamma) & \to & K[v_i], (h,\gamma) & (i \in \{1,2\}) \\
K[\mathsf{if}\, n\, e_1\, e_2], (h,\gamma) & \to & K[e_1], (h,\gamma) & (\text{if } n > 0) \\
K[\mathsf{if}\, 0\, e_1\, e_2], (h,\gamma) & \to & K[e_2], (h,\gamma) & \\
K[\mathsf{ref}\, v], (h,\gamma) & \to & K[l], (h[l \mapsto v], \gamma) & (l \notin \mathsf{dom}(h)) \\
K[!l], (h,\gamma) & \to & K[h(l)], (h,\gamma) & (l \in \mathsf{dom}(h)) \\
K[l := v], (h,\gamma) & \to & K[()], (h[l \mapsto v], \gamma) & (l \in \mathsf{dom}(h)) \\
K[\mathsf{emit}\, v], (h,\gamma) & \xrightarrow{v} & K[()], (h,\gamma) &
\end{array}$$

The above defines a labelled transition system with labels from $\mathit{Val} \cup \{\epsilon\}$ and, by taking its reflexive transitive closure ($\to^*$), we obtain labels from Val^*, which we shall call *traces*.

We next relate the behaviour of each expression with that of its counterpart where all emits have been omitted. We write $\widehat{e}$ for the expression obtained from e by replacing all occurrences of $\mathsf{emit}\, v$ by $()$, and extend this notation to evaluation contexts and functional environments in the expected manner.

Theorem 1. *For all $e, e', h, h', \gamma, \gamma'$, if $\widehat{e}, (h, \widehat{\gamma}) \to^* e', (h', \gamma')$ then there exists a trace t, an expression e'' and an environment γ'' such that $e' = \widehat{e''}, \gamma' = \widehat{\gamma''}$ and $e, (h, \gamma) \xrightarrow{t}{}^* e'', (h', \gamma'')$.*

This theorem justifies the fact that instrumenting terms with emits does not change their semantics. Its proof is in Appendix C.

4 Logic

In this section we introduce our logic and prove some meta-theoretic results. The logic is a standard higher-order separation logic for the λ-calculus introduced in §3, extended with new primitives for reasoning about trace properties and programs that emit traces.

The logic consists of an assertion logic for reasoning about machine states and a specification logic for reasoning about the behaviour of programs. The assertion and specification logics are constructed over the same simply-typed term language. The types of this simply-typed term language are given below:

$$\sigma, \tau \;::=\; 1 \mid \mathsf{Bool} \mid \mathsf{Nat} \mid \tau \to \sigma \mid \tau \times \sigma \mid \mathsf{seq}\ \tau \mid \mathsf{Prop} \mid \mathsf{Spec} \mid \mathsf{Val} \mid \mathsf{Exp} \mid \mathsf{Loc}$$

$$\frac{\Gamma \vdash M : \mathsf{Loc} \qquad \Gamma \vdash N : \mathsf{Val}}{\Gamma \vdash M \mapsto N : \mathsf{Prop}}\ \textsc{TPointsTo} \qquad \frac{\Gamma \vdash P : \mathsf{Prop} \qquad \Gamma \vdash M : \mathsf{Exp} \qquad \Gamma \vdash Q : \mathsf{Val} \to \mathsf{Prop}}{\Gamma \vdash \{P\}\ M\ \{Q\} : \mathsf{Spec}}\ \textsc{THoare}$$

$$\frac{\Gamma \vdash P : \mathsf{Prop}}{\Gamma \vdash \mathsf{valid}(P) : \mathsf{Spec}}\ \textsc{TValid} \qquad \frac{\Gamma \vdash M : \mathsf{Spec}}{\Gamma \vdash \mathsf{spec}(M) : \mathsf{Prop}}\ \textsc{TSpec} \qquad \frac{\Gamma \vdash M : \mathsf{seq}\ \tau \quad \Gamma \vdash N_1 : \sigma \quad \Gamma \vdash N_2 : \sigma \times \tau \to \sigma}{\Gamma \vdash \mathsf{fold}(M, N_1, N_2) : \sigma}\ \textsc{TFold}$$

Fig. 1. Excerpt of typing rules.

The types include standard base types (unit, Booleans, and natural numbers) and type formers for functions, products and finite sequences. Additionally, there is a type for resource assertions (Prop), a type for specifications (Spec), and types for values, expressions and locations from the programming language (Val, Exp and Loc, respectively). The type $\mathsf{seq}\ \tau$ is the type of finite sequences of elements of type τ; we will use this type to represent traces.

The terms of the language are generated by the following grammar. The language includes a simply-typed λ-calculus with pairs, a higher-order logic with equality, primitive operators on finite sequences ($\mathsf{nil}, \mathsf{cons}, \mathsf{fold}$), primitive separation logic resources (emp, $*$, $-\!\!*$, $\mapsto$), Hoare triples and embeddings back and forth between the specification and assertion logic (valid and spec) and three trace primitives (trace, hist and inv).

$$\begin{aligned} P, Q, N, M ::= {} & x \mid \lambda x : \tau.\, M \mid M N \mid () \mid (M, N) \mid \pi_i(M) \mid \bot \mid \top \mid P \vee Q \mid P \wedge Q \\ & \mid P \implies Q \mid \forall x : \tau.\, P \mid \exists x : \tau.\, P \mid M =_\tau N \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\ M\ \mathsf{then}\ N_1\ \mathsf{else}\ N_1 \\ & \mid \mathsf{valid}(P) \mid \mathsf{emp} \mid P * Q \mid P -\!\!* Q \mid M \mapsto N \mid \{P\}\ e\ \{Q\} \mid \mathsf{spec}(M) \mid \mathsf{nil}_\tau \\ & \mid \mathsf{cons}(M, N) \mid \mathsf{fold}(M, N_1, N_2) \mid \mathsf{trace}(M) \mid \mathsf{hist}(M) \mid \mathsf{inv}(M) \end{aligned}$$

The typing rules for the λ-calculus part are standard and have been omitted. Figure 1 includes an excerpt of some of the more interesting typing rules.

The typing judgement has the form $\Gamma \vdash M : \tau$ where Γ is a term context associating types with variables. Note that the typing rule for Hoare triples (THoare) takes an assertion P as a precondition and a predicate Q as a postcondition to allow the postcondition to refer to the return value of the computation. Since we are reasoning about a λ-calculus (so that variables are immutable), we do not distinguish program and logical variables. The expression e in a Hoare triple is thus typed in the same context as the Hoare triple, allowing us to refer to logical variables as program variables.

Judgements of the assertion logic have the form $\Gamma \mid \Theta \mid P \vdash Q$, where Γ is again a term context, associating types to variables, Θ is context of assumed specifications, and P and Q are resource assertions. The judgement should be interpreted as: under the assumption the specifications Θ hold, the resource assertion P entails the resource assertion Q. The assertion logic consists of the usual entailment rules for higher-order separation logic [5] with the following additions,

$$\textsc{PSpec}\quad \Gamma \mid \Theta, S \mid P \vdash \mathsf{spec}(S) \qquad\qquad \textsc{PValid}\quad \Gamma \mid \Theta, \mathsf{valid}(Q) \mid P \vdash Q$$

$$\textsc{Hyp}\quad \Gamma \mid \Theta, S \vdash S \qquad \textsc{Ret}\quad \Gamma, v : \mathsf{Val} \mid \Theta \vdash \{\top\}\ v\ \{r.\ r = v\} \qquad \frac{\Gamma \mid \Theta \mid - \vdash P}{\Gamma \mid \Theta \vdash \mathsf{valid}(P)}\ \textsc{Valid}$$

$$\frac{\Gamma \mid \Theta, S \vdash \{P\}\ e\ \{R\}}{\Gamma \mid \Theta \vdash \{P * \mathsf{spec}(S)\}\ e\ \{R\}}\ \textsc{SpecOut} \qquad \frac{\Gamma \mid \Theta \vdash \{P\}\ e\ \{Q\}}{\Gamma \mid \Theta \vdash \{P * R\}\ e\ \{r.\ Q(r) * R\}}\ \textsc{Frame}$$

$$\frac{\Gamma \mid \Theta \vdash \{P\}\ e\ \{x. Q\} \qquad \Gamma, x : \mathsf{Val} \mid \Theta \vdash \{Q\}\ K[x]\ \{r. R\} \qquad x \notin FV(\Theta)}{\Gamma \mid \Theta \vdash \{P\}\ K[e]\ \{r. \exists x : \mathsf{Val}. R\}}\ \textsc{Bind}$$

$$\frac{\Gamma \mid \Theta \mid P_1 \vdash P_2 \qquad \Gamma \mid \Theta \vdash \{P_2\}\ e\ \{Q_2\} \qquad \Gamma, x : \mathsf{Val} \mid \Theta \mid Q_2(x) \vdash Q_1(x)}{\Gamma \mid \Theta \vdash \{P_1\}\ e\ \{Q_1\}}\ \textsc{Csq}$$

$$\frac{\Gamma, x : \mathsf{Val} \mid \Theta \vdash \{P\}\ e\ \{Q\} \qquad x \notin FV(\Theta)}{\Gamma \mid \Theta \vdash \{\mathsf{emp}\}\ \lambda x. e\ \{r. \forall x : \mathsf{Val}. \mathsf{spec}(\{P\}\ rx\ \{Q\})\}}\ \textsc{Abs}$$

$$\textsc{Write}\quad \Gamma, v, l : \mathsf{Val} \mid \Theta \vdash \{l \mapsto \_\}\ l := v\ \{r.\ l \mapsto v * r = ()\} \qquad \textsc{PInvDupl}\quad \Gamma \mid \Theta \mid \mathsf{inv}(I) \vdash \mathsf{inv}(I) * \mathsf{inv}(I)$$

$$\textsc{Read}\quad \Gamma, v, l : \mathsf{Val} \mid \Theta \vdash \{l \mapsto v\}\ !l\ \{r.\ l \mapsto v * r = v\} \qquad \textsc{Alloc}\quad \Gamma, v : \mathsf{Val} \mid \Theta \vdash \{\mathsf{emp}\}\ \mathsf{ref}\, v\ \{r. r \mapsto v\}$$

$$\textsc{PHistDupl}\quad \Gamma \mid \Theta \mid \mathsf{hist}(t) \vdash \mathsf{hist}(t) * \mathsf{hist}(t) \qquad \textsc{PUseHist}\quad \Gamma \mid \Theta \mid \mathsf{trace}(t_1) * \mathsf{hist}(t_2) \vdash \mathsf{trace}(t_1) * t_2 \leq_{pref} t_1$$

$$\textsc{PAllocHist}\quad \Gamma \mid \Theta \mid \mathsf{trace}(t) \vdash \mathsf{trace}(t) * \mathsf{hist}(t) \qquad \frac{\Gamma \mid \Theta \vdash I(t \cdot v)}{\Gamma \mid \Theta \vdash \{\mathsf{trace}(t) * \mathsf{inv}(I)\}\ \mathsf{emit}\, v\ \{\mathsf{trace}(t \cdot v)\}}\ \textsc{Emit}$$

Fig. 2. Selected specification and assertion entailments.

which allow us to use the specification context in propositional entailments.

Judgements of the specification logic have the form $\Gamma \mid \Theta \vdash S$, where again Γ and Θ are term and specification contexts respectively, and S is a specification The specification logic consists of the usual entailment rules for higher-order logic, with the addition of the rules in Figure 2. As a notational convention, we drop the λ in the postcondition: $\{P\}\ e\ \{r. Q\}$ means $\{P\}\ e\ \{\lambda r : \mathsf{Val}. Q\}$, and $\{P\}\ e\ \{Q\}$ stands for $\{P\}\ e\ \{\lambda\_ : \mathsf{Val}. Q\}$.

Example 1. Consider the specification $\Phi(P_{init}, bracket)$ defined as follows:

$$\exists \mathsf{inv} : \mathsf{Prop}. \mathsf{valid}(P_{init} \implies \mathsf{inv}) \land \forall P, Q : \mathsf{Prop}. \forall f : \mathsf{Val}.$$
$$\{\mathsf{inv} * P * \mathsf{spec}(\{P\}\ f()\ \{Q\})\}\ bracket(f)\ \{\mathsf{inv} * Q\}$$

This specification describes a module which requires some initial resource P_{init} to establish its invariant inv. The assertion $P_{init} \implies \mathsf{inv}$ permits the invariant to be constructed; since this is a resource assertion, it is wrapped in valid to produce the specification which asserts that the implication holds for all resources. The module provides one function, *bracket*, that is specified by the Hoare triple. The specification states that, when *bracket* is applied to a function f that takes precondition P to postcondition Q, then it will behave similarly in the presence of the invariant inv. The condition on the argument f is specified with a nested triple in the precondition; since a triple is a Spec and the precondition must be a Prop, the triple must be wrapped by spec.

As a trivial implementation, we can prove $\text{-}|\text{-} \vdash \Phi(\mathsf{emp}, \lambda f.\, f())$. A client specification S may be proved against an abstract module by proving:

$$P_{init} : \mathsf{Prop}, \mathit{bracket} : \mathsf{Val} \mid \Phi(P_{init}, \mathit{bracket}) \vdash S$$

This can then be composed with the module implementation. □

Note that the specification logic lacks an application rule for applying an argument to a function. Instead, the abstraction rule returns a specification for the given function, applied to an arbitrary argument x. To use this specification, we instantiate the specification with the actual function argument and pull out the nested triple to the context using the SpecOut rule.

Trace primitives. The logic includes three basic assertions, $\mathsf{trace}(t)$, $\mathsf{hist}(t)$ and $\mathsf{inv}(I)$, for reasoning about traces. The typing rules for these primitive trace assertions are given below. We use Trace as shorthand for $\mathsf{seq}\,\mathsf{Val}$.

$$\frac{\Gamma \vdash t : \mathsf{Trace}}{\Gamma \vdash \mathsf{trace}(t) : \mathsf{Prop}}\ \text{TTrace} \qquad \frac{\Gamma \vdash t : \mathsf{Trace}}{\Gamma \vdash \mathsf{hist}(t) : \mathsf{Prop}}\ \text{THist} \qquad \frac{\Gamma \vdash I : \mathsf{Trace} \to \mathsf{Bool}}{\Gamma \vdash \mathsf{inv}(I) : \mathsf{Prop}}\ \text{TInv}$$

The trace 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. The trace t is represented as a finite sequence of values. Since $\mathsf{trace}(t)$ asserts exclusive right to emit events, it cannot be duplicated. The trace resource thus 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 expresiveness, 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 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 captured in the axioms PHistDupl, PAllocHist and PUseHist rules of the extended logic, given in Figure 2.

For several of our examples we require that all traces generated belong to a restricted language of traces. We express this formally using the invariant resource, $\mathsf{inv}(I)$, which defines a trace invariant that everyone must obey. Here I is a set of traces and $\mathsf{inv}(I)$ asserts that the current trace invariant is given by I. Since it specifies an invariant, the $\mathsf{inv}(I)$ resource is duplicable (axiom PInvDupl in Figure 2). A triple $\{P\}\ e\ \{Q\}$ expresses that for every initial state satisfying

P if e executes to a terminal state then the terminal state satisfies Q and the current trace at every intermediate state (including the initial and terminal state) satisfies the trace invariant.

To emit an event v we thus require ownership of the trace resource $\mathsf{trace}(t)$ and that the current trace after emitting the event satisfies the trace invariant I. The proof rule Emit in Figure 2 captures this. Note that the $t \cdot v \in I$ assumption in the Emit rule is not strictly part of the syntax of our term language, but may be defined using fold. The same is true of $\leq_{pref}$ and other operations on finite sequences, such as concatenation, length of a sequence and a subsequence operation. In subsequent sections we will use these definable operations without further mention. We will also occasionally need inductively defined predicates on traces. Since we are working in a higher-order logic, such predicates are definable within the logic using the usual impredicative Knaster-Tarski definition of least-fixed points of monotone operators.

5 Proving Trace Properties

In this section we show how to derive trace properties about client-library interactions from the library's specification. We demonstrate our approach through a series of increasingly complex examples, starting with the basic file library example from the Introduction.

The basic idea is to prove that for any library implementation satisfying the abstract library specification, the wrapped library implementation satisfies the same abstract specification and moreover the traces generated by the wrapping satisfy a given invariant. This is achieved by reinterpreting the abstract representation predicates of the specifications to additionally relate the abstract state with the current trace using trace assertions. 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 2 below formalises this idea. Here I_{init} is an initialiser operation to initialize the internal state P_0 of the library. Theorem 2 allows us to prove that traces generated by running a client e linked with a library implementation I_{ops}, after running the initialiser, belong to a given language $\mathcal{L}$.

Theorem 2. *Given any specification Φ, resource P_0, initialiser I_{init}, library implementation I_{ops}, machine states s, s', trace t, clients e, e' and language $\mathcal{L}$, if the following conditions hold then $t \in \mathcal{L}$:*

- $- \vdash \Phi : \mathsf{Prop} \times \mathsf{Exp} \to \mathsf{Spec}$ *and* $- \vdash P_0 : \mathsf{Prop}$
- $- \mid - \vdash \forall ops, P.\ \Phi(P, ops) \implies \{P\}\ e\ \{\top\}$
- $- \mid - \vdash \{\top\}\ I_{init}\ \{P_0\}$
- $- \mid - \vdash \Phi(P_0 * \mathsf{trace}(\varepsilon) * \mathsf{inv}(\mathcal{L}), I_{ops})$
- $(I_{init}; e[I_{ops}/ops]), s \xrightarrow{t}{}^{*} e', s' \not\to$

Proof. Follows from soundness of the logic (Lemma 1 in §6). □

The above theorem requires that we provide a library implementation that satisfies the abstract specification Φ and generates traces in the language $\mathcal{L}$. To use this theorem to derive a trace property from the 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 Φ then so does the wrapped version $\mathbf{wrap}(M)$ *and*, additionally, the wrapped version generates traces in the $\mathcal{L}$ language: $P_0 : \mathsf{Prop}, ops : \mathsf{Exp} \mid \Phi(P_0, ops) \vdash \Phi(P_0 * \mathsf{trace}(\varepsilon) * \mathsf{inv}(\mathcal{L}), \mathbf{wrap}(ops))$.

5.1 File Library

To illustrate the idea, we begin by recalling the file library example from the Introduction. We prove that the associated separation logic specification enforces that clients verified against the specification only close and read when the file is open. To capture this property, we define a wrapper function $\mathbf{wrap}_{\mathsf{file}}$ that instruments an implementation of the file library to emit events about calls to `open`, `close` and `read`. Formally, we take a file library to be a triple consisting of an `open`, a `close` and a `read` function.

$$\mathbf{wrap}_{\mathsf{file}} \stackrel{\text{def}}{=} \lambda(open, close, read).\\ \big(\lambda\_.\, open(); \mathsf{emit}\,\texttt{open},\ \lambda\_.\, close(); \mathsf{emit}\,\texttt{close},\ \lambda\_.\, read(); \mathsf{emit}\,\texttt{read}\big)$$

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

$$\mathsf{file}_{\mathsf{trace}}(t) \stackrel{\text{def}}{=} \forall n.\, t[n] = \texttt{read} \vee t[n] = \texttt{close} \implies \mathsf{isopen}(t, n)$$

$$\mathsf{noclose}(t, n, m) \stackrel{\text{def}}{=} \forall k.\, n < k < m \implies t[k] \neq \texttt{close}$$

$$\mathsf{isopen}(t, n) \stackrel{\text{def}}{=} \exists m < n.\, t[m] = \texttt{open} \wedge \mathsf{noclose}(t, m, n)$$

The valid file trace predicate, $\mathsf{file}_{\mathsf{trace}}(t)$, expresses that the trace t only contains read and close events when the file is open. Recall the SL specification of the file library from the Introduction, here written more formally.

$$\Phi_{\mathsf{file}}(P_0, (open, close, read)) \stackrel{\text{def}}{=} \exists \mathsf{open}, \mathsf{closed} : \mathsf{Prop}.\, \mathsf{valid}(P_0 \implies \mathsf{closed}) \wedge\\ \{\mathsf{closed}\}\ open()\ \{\mathsf{open}\} \wedge \{\mathsf{open}\}\ close()\ \{\mathsf{closed}\} \wedge \{\mathsf{open}\}\ read()\ \{\mathsf{open}\}$$

To prove the specification enforces the intended protocol, we proceed by proving that the wrapping preserves satisfaction of the specification and generates traces in $\mathcal{L}_{\mathsf{file}}$.

Lemma 1. $P_0, ops \mid \Phi_{\mathit{file}}(P_0, ops) \vdash \Phi_{\mathit{file}}(P_0 * \mathsf{trace}(\varepsilon) * \mathsf{inv}(\mathcal{L}_{\mathsf{file}}), \mathbf{wrap}_{\mathit{file}}(ops))$.

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, $\mathsf{open}_{\mathsf{w}}$, should express that the current trace t is in $\mathcal{L}_{\mathsf{file}}$ and that the trace t is open. Likewise, the wrapped closed resource, $\mathsf{closed}_{\mathsf{w}}$ should simply express that the current trace is in $\mathcal{L}_{\mathsf{file}}$.

$$\mathsf{open}_{\mathsf{w}} \stackrel{\text{def}}{=} \mathsf{open} * \exists t \in \mathcal{L}_{\mathsf{file}}.\, \mathsf{trace}(t) * \mathsf{inv}(\mathcal{L}_{\mathsf{file}}) * \mathsf{isopen}(t, |t| + 1)$$

$$\mathsf{closed}_{\mathsf{w}} \stackrel{\text{def}}{=} \mathsf{closed} * \exists t \in \mathcal{L}_{\mathsf{file}}.\, \mathsf{trace}(t) * \mathsf{inv}(\mathcal{L}_{\mathsf{file}})$$

We need to prove that $P_0 * \mathsf{trace}(\varepsilon) * \mathsf{inv}(\mathcal{L}_{\mathsf{file}}) \implies \mathsf{closed}_{\mathsf{w}}$ assuming $P_0 \implies \mathsf{closed}$, which follows trivially from the definition of $\mathsf{closed}_{\mathsf{w}}$. It remains to prove that the wrapped methods satisfy their specifications. Below we give a proof outline for the wrapped `open` method.

$\{\mathsf{closed}_{\mathsf{w}}\}$
$\{\mathsf{closed} * \exists t \in \mathcal{L}_{\mathsf{file}}.\, \mathsf{trace}(t) * \mathsf{inv}(\mathcal{L}_{\mathsf{file}})\}$
$\mathsf{open}();$
$\{\mathsf{open} * \exists t \in \mathcal{L}_{\mathsf{file}}.\, \mathsf{trace}(t) * \mathsf{inv}(\mathcal{L}_{\mathsf{file}})\}$
$\textbf{emit}\ \mathsf{open};$
$\{\mathsf{open} * \exists t \in \mathcal{L}_{\mathsf{file}}.\, \mathsf{trace}(t) * \mathsf{inv}(\mathcal{L}_{\mathsf{file}}) * \mathsf{isopen}(t, |t|+1)\}$
$\{\mathsf{open}_{\mathsf{w}}\}$

Here we use the assumed specification of the underlying `open` method to verify the call to open and we used the following property of $\mathcal{L}_{\mathsf{file}}$ and isopen to prove that emitting `open` would result in a trace in $\mathcal{L}_{\mathsf{file}}$ that was open.

$$\forall t \in \mathcal{L}_{\mathsf{file}}.\, t \cdot \mathtt{open} \in \mathcal{L}_{\mathsf{file}} \wedge \mathsf{isopen}(t \cdot \mathtt{open}, |t \cdot \mathtt{open}| + 1)$$

The proof outlines for the `close` and `read` operations are similar, but use the following property to justify the emits.

$$\forall t \in \mathcal{L}_{\mathsf{file}}.\, \mathsf{isopen}(t, |t|+1) \implies t \cdot \mathtt{close} \in \mathcal{L}_{\mathsf{file}} \wedge t \cdot \mathtt{read} \in \mathcal{L}_{\mathsf{file}} \quad \square$$

5.2 Iterators on Collections

We consider a collections library that provides methods for modifying a collection as well as iterating over it. To ensure a well-defined semantics for iterators, we require the following property:

> *An iterator over a collection should 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 collections library and clients. To capture it formally, we first define a suitable wrapper for the library that produces appropriate trace events. We take a collections library to be a tuple consisting of five operations: `size`, `add`, `remove`, `iterator` and `next`. 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 from the collection. Finally, `iterator` returns a new iterator for the collection, while `next` returns the next element of a given iterator.

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:

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

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 $\mathcal{L}_{\mathsf{coll}}$ of safe behaviours contains all strings $t \in \Sigma^*$ such that, for all $1 \le i \le |t|$:

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 ℓ must be preceded by a call to `iterator`() which returns ℓ. In addition, there should be no modification of the collection between those two events.

To enforce this protocol we use a cut-down version of the iterator specification in [19] that does not track the contents of the underlying collections.

$$\begin{aligned}
&\Phi_{\mathsf{coll}}(P_0, (\mathit{size}, \mathit{add}, \mathit{remove}, \mathit{iterator}, \mathit{next})) \stackrel{\text{def}}{=} \\
&\qquad \exists \mathsf{coll} : \mathsf{Val} \to \mathsf{Prop}.\, \exists \mathsf{iter} : \mathsf{Val} \times \mathsf{Val} \to \mathsf{Prop}.\mathsf{valid}(P_0 \implies \exists c : \mathsf{Val}.\ \mathsf{coll}(c)) \ \wedge \\
&\qquad\quad \forall c : \mathsf{Val}.\ \{\mathsf{coll}(c)\}\ \mathit{size}()\ \{\mathsf{coll}(c)\}\ \wedge \\
&\qquad\quad \forall c, x : \mathsf{Val}.\ \{\mathsf{coll}(c)\}\ \mathit{add}(x)\ \{\exists c' : \mathsf{Val}.\, \mathsf{coll}(c')\}\ \wedge \\
&\qquad\quad \forall c : \mathsf{Val}.\ \{\mathsf{coll}(c)\}\ \mathit{remove}(x)\ \{\exists c' : \mathsf{Val}.\, \mathsf{coll}(c')\}\ \wedge \\
&\qquad\quad \forall c : \mathsf{Val}.\ \{\mathsf{coll}(c)\}\ \mathit{iterator}()\ \{r.\ \mathsf{coll}(c) * \mathsf{iter}(r, c)\}\ \wedge \\
&\qquad\quad \forall c, x : \mathsf{Val}.\ \{\mathsf{coll}(c) * \mathsf{iter}(x, c)\}\ \mathit{next}(x)\ \{\mathsf{coll}(c) * \mathsf{iter}(x, c)\}
\end{aligned}$$

The specification introduces two types of resources, coll and iter, to formally capture the protocol. The $\mathsf{coll}(c)$ resource is indexed by an abstract version number c while the $\mathsf{iter}(p, c)$ resource expresses that p is an iterator and the version number of the underlying collection was c when the iterator was created. The operations that destructively update the collection consume a $\mathsf{coll}(c)$ resource and produce a $\mathsf{coll}(c')$ resource for an existentially quantified version number c'. As a result, after a destructive update, we should no longer be able to satisfy the precondition of `next`, as it requires ownership of a coll resource and an iter resource with a common version number c.

To establish this intuition formally we 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 2. $P_0, \mathit{ops} \mid \Phi_{\mathit{coll}}(P_0, \mathit{ops}) \vdash \Phi_{\mathit{coll}}(P_0 * \mathsf{trace}(\varepsilon) * \mathsf{inv}(\mathcal{L}_{\mathit{coll}}), \mathbf{wrap}_{\mathit{coll}}(\mathit{ops}))$.

Proof (sketch). To prove that the wrapped implementation satisfies the collections specification and produces traces in $\mathcal{L}_{\mathsf{coll}}$, 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 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 $\mathsf{coll_w}$ and $\mathsf{iter_w}$ as follows:

$$\begin{aligned}\mathsf{coll_w}(x) &\stackrel{\text{def}}{=} \exists y : \mathsf{Val}.\, \exists n : \mathsf{Nat}.\, x = (y, n) * \mathsf{coll}(y) * \\ &\qquad \exists t \in \mathcal{L}_{\mathsf{coll}}.\, \mathsf{trace}(t) * \mathsf{inv}(\mathcal{L}_{\mathsf{coll}}) * \mathtt{add}, \mathtt{remove} \notin t[n+1..] * n \leq |t| \\ \mathsf{iter_w}(r, x) &\stackrel{\text{def}}{=} \exists y : \mathsf{Val}.\, \exists n : \mathsf{Nat}.\, x = (y, n) * \mathsf{iter}(y) * \\ &\qquad \exists t \in \mathcal{L}_{\mathsf{coll}}.\, \mathsf{hist}(t) * \langle \mathtt{iterator}, r \rangle \in t[n+1..]\end{aligned}$$

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: $P_0 * \mathsf{inv}(\mathcal{L}_{\mathsf{coll}}) * \mathsf{trace}(\varepsilon) \implies \exists c : \mathsf{Val}.\, \mathsf{coll_w}(c)$. This follows easily from the $P_0 \implies \exists c' : \mathsf{Val}.\, \mathsf{coll}(c')$ 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 specification. 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`.

[Context $c, \mathtt{z} : \mathsf{Val}$]
$\{\mathsf{coll_w}(c)\}$
$\{\exists x, n, t.\, c = (x, n) * \mathsf{coll}(x) * \mathsf{trace}(t) * \mathsf{inv}(\mathcal{L}_{\mathsf{coll}}) * t \in \mathcal{L}_{\mathsf{coll}}$
$\quad * \mathtt{add}, \mathtt{remove} \notin t[n+1..] * n \leq |t|\}$
add(z);
$\{\exists x', n, t.\, \mathsf{coll}(x') * \mathsf{trace}(t) * \mathsf{inv}(\mathcal{L}_{\mathsf{coll}}) * t \in \mathcal{L}_{\mathsf{coll}} * \mathtt{add}, \mathtt{remove} \notin t[n+1..] * n \leq |t|\}$
emit add;
$\{\exists x', n, t.\, \mathsf{coll}(x') * \mathsf{trace}(t \cdot \mathtt{add}) * \mathsf{inv}(\mathcal{L}_{\mathsf{coll}}) * t \in \mathcal{L}_{\mathsf{coll}}$
$\quad * \mathtt{add}, \mathtt{remove} \notin t[n+1..] * n \leq |t|\}$
$\{\exists c' : \mathsf{Val}.\, \mathsf{coll_w}(c')\}$

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

$$\forall x', n, t. \begin{pmatrix} \mathsf{coll}(x') * \mathsf{trace}(t \cdot \mathtt{add}) * \mathsf{inv}(\mathcal{L}_{\mathsf{coll}}) * t \in \mathcal{L}_{\mathsf{coll}} \\ * \, n \leq |t| * \mathtt{add}, \mathtt{remove} \notin t[n+1..] \end{pmatrix} \implies \exists c' : \mathsf{Val}.\, \mathsf{coll}_{\mathsf{w}}(c')$$

This follows easily by taking c' to be $(x', |t \cdot \mathtt{add}|)$, as $t \cdot \mathtt{add}$ contains no `add` or `remove` events after the $|t \cdot \mathtt{add}|$-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. Below we give a proof outline for the `iterator` method:

[Context c : Val]
$\{\mathsf{coll}_{\mathsf{w}}(c)\}$
$\{\exists x, n, t.\, c = (x, n) * \mathsf{coll}(x) * \mathsf{trace}(t) * \mathsf{inv}(\mathcal{L}_{\mathsf{coll}}) * t \in \mathcal{L}_{\mathsf{coll}}$
$\quad * \, \mathtt{add}, \mathtt{remove} \notin t[n+1..] * n \leq |t|\}$
let r = iterator() **in**
$\{\exists x, n, t.\, c = (x, n) * \mathsf{coll}(x) * \mathsf{iter}(\mathtt{r}, x) * \mathsf{trace}(t)$
$\quad * \, \mathsf{inv}(\mathcal{L}_{\mathsf{coll}}) * t \in \mathcal{L}_{\mathsf{coll}} * \mathtt{add}, \mathtt{remove} \notin t[n+1..] * n \leq |t|\}$
emit⟨iterator, r⟩;
$\{\exists x, n, t.\, c = (x, n) * \mathsf{coll}(x) * \mathsf{iter}(\mathtt{r}, x) * \mathsf{trace}(t \cdot \langle \mathtt{iterator}, \mathtt{r} \rangle)$
$\quad * \, \mathsf{inv}(\mathcal{L}_{\mathsf{coll}}) * t \in \mathcal{L}_{\mathsf{coll}} * \mathtt{add}, \mathtt{remove} \notin t[n+1..] * n \leq |t|\}$
$\{\mathsf{coll}_{\mathsf{w}}(c) * \mathsf{iter}_{\mathsf{w}}(\mathtt{r}, c)\}$
r
$\{r.\, \mathsf{coll}_{\mathsf{w}}(c) * \mathsf{iter}_{\mathsf{w}}(r, c)\}$

As before, we are left with two proof obligations: $t \cdot \langle \mathtt{iterator}, \mathtt{r} \rangle \in \mathcal{L}_{\mathsf{coll}}$ and:

$$\forall x, n, t.\, (\, c = (x, n) * \mathsf{coll}(x) * \mathsf{iter}(\mathtt{r}, x) * \mathsf{trace}(t \cdot \langle \mathtt{iterator}, \mathtt{r} \rangle) * \mathsf{inv}(\mathcal{L}_{\mathsf{coll}}) \\ * \, t \in \mathcal{L}_{\mathsf{coll}} * \mathtt{add}, \mathtt{remove} \notin t[n+1..] * n \leq |t|) \implies \mathsf{coll}_{\mathsf{w}}(c) * \mathsf{iter}_{\mathsf{w}}(\mathtt{r}, c)$$

To discharge this last proof obligation, we use the PAllocHist rule to introduce a history resource $\mathsf{hist}(t \cdot \langle \mathtt{iterator}, \mathtt{r} \rangle)$ and since $n \leq |t|$ it follows that $\langle \mathtt{iterator}, \mathtt{r} \rangle \in (t \cdot \langle \mathtt{iterator}, \mathtt{r} \rangle)[n+1..]$, as required by $\mathsf{iter}_{\mathsf{w}}(\mathtt{r}, c)$.

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 an argument `x`:

[Context $c, \mathtt{x}$: Val]
$\{\mathsf{coll}_{\mathsf{w}}(c) * \mathsf{iter}_{\mathsf{w}}(\mathtt{x}, c)\}$
$\{\exists y, n, t, t'.\, c = (y, n) * \mathsf{coll}(y) * \mathsf{iter}(\mathtt{x}, c) * \mathsf{trace}(t) * \mathsf{hist}(t') * \mathsf{inv}(\mathcal{L}_{\mathsf{coll}})$
$\quad * \, t, t' \in \mathcal{L}_{\mathsf{coll}} * \mathtt{add}, \mathtt{remove} \notin t[n+1..] * n \leq |t| * \langle \mathtt{iterator}, \mathtt{x} \rangle \in t'[n+1..]\}$
let r = next(x);
$\{\exists y, n, t, t'.\, c = (y, n) * \mathsf{coll}(y) * \mathsf{iter}(\mathtt{x}, c) * \mathsf{trace}(t) * \mathsf{hist}(t') * \mathsf{inv}(\mathcal{L}_{\mathsf{coll}})$
$\quad * \, t, t' \in \mathcal{L}_{\mathsf{coll}} * \mathtt{add}, \mathtt{remove} \notin t[n+1..] * n \leq |t| * \langle \mathtt{iterator}, \mathtt{x} \rangle \in t'[n+1..]\}$
emit⟨next, x⟩;
$\{\exists y, n, t, t'.\, c = (y, n) * \mathsf{coll}(y) * \mathsf{iter}(\mathtt{x}, c) * \mathsf{trace}(t \cdot \langle \mathtt{next}, \mathtt{x} \rangle) * \mathsf{hist}(t') * \mathsf{inv}(\mathcal{L}_{\mathsf{coll}})$
$\quad * \, t, t' \in \mathcal{L}_{\mathsf{coll}} * \mathtt{add}, \mathtt{remove} \notin t[n+1..] * n \leq |t| * \langle \mathtt{iterator}, \mathtt{x} \rangle \in t'[n+1..]\}$
$\{\mathsf{coll}_{\mathsf{w}}(c) * \mathsf{iter}_{\mathsf{w}}(\mathtt{x}, c)\}$
r
$\{\mathsf{coll}_{\mathsf{w}}(c) * \mathsf{iter}_{\mathsf{w}}(\mathtt{x}, c)\}$

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

$$t \in \mathcal{L}_{\mathsf{coll}} \wedge \texttt{add}, \texttt{remove} \notin t[n{+}1..] \wedge \langle \texttt{iterator}, \texttt{x} \rangle \in t[n{+}1..] \Longrightarrow t \cdot \langle \texttt{next}, \texttt{x} \rangle \in \mathcal{L}_{\mathsf{coll}}$$

To apply this property we use PUseHist to conclude from $\mathsf{trace}(t) * \mathsf{hist}(t')$ that $t' \leq_{pref} t$ and thus that $\langle \texttt{iterator}, \texttt{x} \rangle \in t'[n{+}1..] \implies \langle \texttt{iterator}, \texttt{x} \rangle \in t[n{+}1..]$.

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 wish to ensure that 1) clients only access the resource after they have acquired it, and 2) clients do not try to acquire resources they already hold.

To express this property formally, we first define a wrapping function that instruments the library to emit events about the interaction between client and library. Formally, we take a library implementation to be a tuple consisting of a withRes function and an operation op. Below we define a wrapping function for such a library that emits call and return events for all calls where control passes between client and library.

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

We can now state the desired property as a well-bracketing property of the traces generated by the instrumented library. Let us define $\mathcal{L}_{\mathsf{brac}} \subseteq \mathit{Val}^*$ as the prefix closure of the language of all strings s that are of the form:

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

for some $f \in \mathit{Fun}$, $s_{\mathsf{op}} \in (\langle \texttt{call}, \texttt{op} \rangle \cdot \langle \texttt{ret}, \texttt{op} \rangle)^*$ and $s' \in \mathcal{L}_{\mathsf{brac}}$. That is, the strings in $\mathcal{L}_{\mathsf{brac}}$ are well-bracketed sequences of events formed of subsequences adhering to the pattern $\langle \texttt{call}, \texttt{withRes}, f \rangle \cdot \langle \texttt{call}, f \rangle \cdot s_{\mathsf{op}} \cdot \langle \texttt{ret}, f \rangle \cdot \langle \texttt{ret}, \texttt{withRes}, f \rangle$, and subsequences thereof, where s_{op} a sequence of consecutive calls and returns of op.

This trace property is enforced by the following separation logic specification.

$$\begin{aligned}
&\Phi_{\mathsf{brac}}(P_0, (\mathtt{withRes}, \mathtt{op})) \stackrel{\text{def}}{=} \\
&\quad \exists \mathsf{locked} : \mathsf{Prop}.\, \exists \mathsf{unlocked} : \mathsf{Val} \to \mathsf{Prop}.\, \mathsf{valid}(P_0 \Longrightarrow \mathsf{locked}) \wedge \\
&\quad\quad \forall P, Q : \mathsf{Prop}.\, \forall f : \mathsf{Val}. \\
&\quad\quad\quad \{\mathsf{locked} * P * S(P, Q, \mathsf{unlocked}, f)\}\mathtt{withRes}(f)\{\mathsf{locked} * Q\} \wedge \\
&\quad\quad\quad \forall x, y : \mathsf{Val}.\, \{\mathsf{unlocked}(y)\}\ \mathtt{op}(x)\ \{\mathsf{unlocked}(y)\}
\end{aligned}$$

$$S(P, Q, \mathsf{unlocked}, f) \stackrel{\text{def}}{=} \forall y, x : \mathsf{Val}.\, \mathsf{spec}(\{\mathsf{unlocked}(y) * P\}\ f(x)\ \{\mathsf{unlocked}(y) * Q\})$$

This uses two abstract resources, unlocked and locked to capture the well-bracketing aspect of the protocol. In particular, 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 itself call `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`. This specification ensures that `withRes` is forced to call the function provided by the client *exactly* once, as it is required to transform the abstract resource P into Q and the only way it can achieve this is by calling the function provided by the client.

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 $\mathbf{wrap}_{\mathsf{brac}}(M)$ also satisfies the specification and the traces generated by the wrapped implementation are in $\mathcal{L}_{\mathsf{brac}}$.

Lemma 3. *$P_0, ops \mid \Phi_{brac}(P_0, ops) \vdash \Phi_{brac}(P_0 * \mathsf{trace}(\varepsilon) * \mathsf{inv}(\mathcal{L}_{\mathsf{brac}}), \mathbf{wrap}_{brac}(ops))$.*

Proof (Proof sketch). To prove that the wrapped version satisfies the specification and produces traces in $\mathcal{L}_{\mathsf{brac}}$, we first need to define wrapped versions of the abstract representation predicates. The idea is to let the wrapped locked resource, $\mathsf{locked}_{\mathsf{w}}$, express that the current trace t is in $\mathcal{L}_{\mathsf{brac}}$ 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, $\mathsf{unlocked}_{\mathsf{w}}(x)$, 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 $\mathcal{O} = (\langle \mathtt{call}, \mathtt{op}\rangle \cdot \langle \mathtt{ret}, \mathtt{op}\rangle)^*$. $T_1(f)$ 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 $s' \in \mathcal{O}$. 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 $s' \in \mathcal{O}$.

$$\begin{aligned}
T_0 &= \exists t \in \mathcal{L}_{\mathsf{brac}}.\, \mathsf{trace}(t) * \mathsf{inv}(\mathcal{L}_{\mathsf{brac}}) * (|t| > 0 \implies \exists f.\, t[|t|] = \langle \mathtt{ret}, \mathtt{withRes}, f\rangle) \\
T_1(f) &= \exists t \in \mathcal{L}_{\mathsf{brac}}.\, \mathsf{trace}(t) * \mathsf{inv}(\mathcal{L}_{\mathsf{brac}}) * t[|t|] = \langle \mathtt{call}, \mathtt{withRes}, f\rangle \\
T_2(f) &= \exists t \in \mathcal{L}_{\mathsf{brac}}.\, \mathsf{trace}(t) * \mathsf{inv}(\mathcal{L}_{\mathsf{brac}}) * \exists n < |t|.\ t[n] = \langle \mathtt{call}, \mathtt{withRes}, f\rangle \\
&\qquad\qquad \wedge t[n+1] = \langle \mathtt{call}, f\rangle \wedge t[n+2..] \in \mathcal{O}
\end{aligned}$$

$$T_3(f) = \exists t \in \mathcal{L}_{\mathsf{brac}}.\,\mathsf{trace}(t) * \mathsf{inv}(\mathcal{L}_{\mathsf{brac}}) * \exists n < |t|.\ t[n] = \langle \mathtt{call}, \mathtt{withRes}, f\rangle$$
$$\wedge\, t[n+1] = \langle \mathtt{call}, f\rangle \wedge t[n+2..(|t|-1)] \in \mathcal{O} \wedge t[|t|] = \langle \mathtt{ret}, f\rangle$$

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

$$\mathsf{unlocked}_\mathsf{w}(x) \stackrel{\text{def}}{=} \exists y, z : \mathsf{Val}.\, x = (y, z) * \mathsf{unlocked}(y) * T_2(z)$$
$$\mathsf{locked}_\mathsf{w} \stackrel{\text{def}}{=} \mathsf{locked} * T_0$$

It follows easily that $P_0 * \mathsf{trace}(\varepsilon) * \mathsf{inv}(\mathcal{L}_{\mathsf{brac}}) \implies \mathsf{locked}_\mathsf{w}$ from $\mathsf{trace}(\varepsilon) * \mathsf{inv}(\mathcal{L}_{\mathsf{brac}}) \implies T_0$ and the assumption $P_0 \implies \mathsf{locked}.3$

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

$$\forall P, Q : \mathsf{Prop}.\, \forall f : \mathsf{Val}.$$
$$\{\mathsf{locked}_\mathsf{w} * P * S(P, Q, \mathsf{unlocked}_\mathsf{w}, f)\}\pi_1(\mathbf{wrap}_{\mathsf{brac}}(\mathtt{withRes}, \mathsf{op}))(f)\ \ \{\mathsf{locked}_\mathsf{w} * Q\}$$

assuming `withRes` satisfies its specification:

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

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

[Context P, Q : Prop, f : Val]
$\{\mathsf{locked}_\mathsf{w} * P * S(P, Q, \mathsf{unlocked}_\mathsf{w}, f)\}$
$\{\mathsf{locked} * T_0 * P * S(P, Q, \mathsf{unlocked}_\mathsf{w}, f)\}$
  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**
$\{\mathsf{locked} * T_1(f) * P * S(P * T_1(f), Q * T_3(f), \mathsf{unlocked}, g)\}$
  withRes(g);
$\{\mathsf{locked} * T_3(f) * Q\}$
  emit⟨ret, withRes, f⟩
$\{\mathsf{locked} * T_0 * Q\}$
$\{\mathsf{locked}_\mathsf{w} * Q\}$

The interesting step is showing that from the assumed specification of f we can derive the desired specification for the instrumented version of f:

$$\forall P, Q : \mathsf{Prop}.\, \forall f : \mathsf{Val}.\, S(P, Q, \mathsf{unlocked}_\mathsf{w}, f)$$
$$\implies S(P * T_1(f), Q * T_3(f), \mathsf{unlocked}, \lambda x.\ \mathsf{emit}\langle \mathtt{call}, f\rangle; f(x); \mathsf{emit}\langle \mathtt{ret}, f\rangle)$$

This follows from the following proof outline

[Context P, Q : Prop and f, x, y : Val]
$\{\mathsf{unlocked}(y) * P * T_1(f) * S(P, Q, \mathsf{unlocked}_\mathsf{w}, f)\}$
  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⟩
$\{\mathsf{unlocked}(y) * Q * T_3(f)\}$

Lastly, we need to show that the instrumented `op` function satisfies its specification. Below we give a proof outline for the instrumented `op` function applied to an argument x:

[Context x, y : Val]
$\{\mathsf{unlocked}_{\mathsf{w}}(y)\}$
$\{\exists a, f.\, y = (a, f) * \mathsf{unlocked}(a) * T_2(f)\}$
  emit⟨call, op⟩;
$\{\exists a, f.\, y = (a, f) * \mathsf{unlocked}(a) * \exists t \in \mathcal{L}_{\mathsf{brac}}.\, \mathsf{trace}(t) * \ \mathsf{inv}(\mathcal{L}_{\mathsf{brac}}) * p(t, f)\}$
  op(x);
$\{\exists a, f.\, y = (a, f) * \mathsf{unlocked}(a) * \exists t \in \mathcal{L}_{\mathsf{brac}}.\, \mathsf{trace}(t) * \ \mathsf{inv}(\mathcal{L}_{\mathsf{brac}}) * p(t, f)\}$
  emit⟨ret, op⟩
$\{\exists a, f.\, y = (a, f) * \mathsf{unlocked}(a) * T_2(f)\}$
$\{\mathsf{unlocked}_{\mathsf{w}}(y)\}$

where $p(t, f) = \exists n < |t|.\ t[n] = \langle \mathtt{call}, \mathtt{withRes}, f\rangle \wedge t[n+1] = \langle \mathtt{call}, f\rangle \wedge t[n+2..] \in (\langle \mathtt{call}, \mathsf{op}\rangle \cdot \langle \mathtt{ret}, \mathsf{op}\rangle)^* \cdot \langle \mathtt{call}, \mathsf{op}\rangle$. ☐

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 topmost 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.

To express this protocol, we first 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`.

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

Let Σ_{st} be the stack alphabet. We can formalize the intended protocol as the language $\mathcal{L}_{\mathsf{stack}}$ defined as the prefix closure of the language of all traces $t \in \mathit{Val}^*$

such that $\mathsf{stk_{tr}}(t, \varepsilon)$ holds, where, given $\alpha \in \Sigma^*_{\mathsf{st}}$, we define $\mathsf{stk_{tr}}(t, \alpha)$ by:

$$
\begin{aligned}
&\mathsf{stk_{tr}}(t, \alpha) \stackrel{\text{def}}{=} (t = \alpha = \varepsilon) \vee \\
&\quad (t = t' \cdot \langle \mathtt{call}, \mathtt{push}, a\rangle \cdot \langle \mathtt{ret}, \mathtt{push}\rangle \wedge \alpha = a :: \alpha' \wedge \mathsf{stk_{tr}}(t', \alpha')) \vee \\
&\quad (t = t' \cdot \langle \mathtt{call}, \mathtt{pop}\rangle \cdot \langle \mathtt{ret}, \mathtt{pop}, ()\rangle \wedge \alpha = \varepsilon \wedge \mathsf{stk_{tr}}(t', \varepsilon)) \vee \\
&\quad (t = t' \cdot \langle \mathtt{call}, \mathtt{pop}\rangle \cdot \langle \mathtt{ret}, \mathtt{pop}, a\rangle \wedge \mathsf{stk_{tr}}(t', a :: \alpha)) \vee \\
&\quad (t = t' \cdot \langle \mathtt{call}, \mathtt{foreach}, f\rangle \cdot t'' \cdot \langle \mathtt{ret}, \mathtt{foreach}\rangle \wedge \mathsf{stk_{tr}}(t', \alpha) \wedge \mathsf{trav}(t'', \alpha, f)) \\
&\mathsf{trav}(t, \alpha, f) \stackrel{\text{def}}{=} (t = \alpha = \varepsilon) \vee \\
&\qquad (t = \langle \mathtt{call}, f, a\rangle \cdot \langle \mathtt{ret}, f\rangle \cdot t' \wedge \alpha = a :: \alpha' \wedge \mathsf{trav}(t', \alpha', f))
\end{aligned}
$$

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

$$
\begin{aligned}
&\Phi(P_{init}, (push, pop, foreach)) \stackrel{\text{def}}{=} \\
&\quad \exists \mathsf{stack} : \mathsf{Val\ seq} \to \mathsf{Prop}.\, \mathsf{valid}(P_{init} \implies \mathsf{stack}(\varepsilon)) \wedge \\
&\quad\ \forall \alpha, a.\, \{\mathsf{stack}(\alpha) \wedge a \neq ()\}\ push(a)\ \{\mathsf{stack}(a :: \alpha)\} \wedge \\
&\quad\ \forall \alpha.\, \{\mathsf{stack}(\alpha)\}\ pop()\ \{r.\ (r = () \wedge \mathsf{stack}(\alpha) \wedge \alpha = \varepsilon) \\
&\qquad\qquad\qquad\qquad \vee\ (\exists \alpha'.\, \alpha = r :: \alpha' \wedge \mathsf{stack}(\alpha'))\} \wedge \\
&\quad\ \forall \alpha, f, I.\ \{\mathsf{stack}(\alpha) * I(\varepsilon) * \forall \beta, a.\, \mathsf{spec}(\{I(\beta)\}\ f(a)\ \{I(a :: \beta)\})\} \\
&\qquad\qquad foreach(f)\{\mathsf{stack}(\alpha) * I(rev(\alpha))\}
\end{aligned}
$$

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 α. 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 α and $I(\alpha)$ is intended to capture the client's state after the function provided by the client has been called on each element of α, in reverse order. This accounts for the $I(rev(\alpha))$ in the post-condition, where rev is the reverse operator on sequences.

Lemma 4. $P_0, ops \mid \Phi(P_0, ops) \vdash \Phi(P_0 * \mathsf{inv}(\mathcal{L}_{\mathsf{stack}}) * \mathsf{trace}(\varepsilon), \mathbf{wrap}_{\mathit{stack}}(ops))$.

Proof (Proof sketch). We proceed by defining a wrapped version of the stack predicate that asserts that the sequence of elements α matches the expected contents of the stack as per the current trace t.

$$\mathsf{stack_w}(\alpha) \stackrel{\text{def}}{=} \mathsf{stack}(\alpha) * \exists t.\, \mathsf{stk_{tr}}(t, \alpha) * \mathsf{trace}(t) * \mathsf{inv}(\mathcal{L}_{\mathsf{stack}})$$

Clearly we have that $P_{init} * \mathsf{inv}(\mathcal{L}_{\mathsf{stack}}) * \mathsf{trace}(\varepsilon) \implies \mathsf{stack_w}(\varepsilon)$ as $\mathsf{stk_{tr}}(\varepsilon, \varepsilon)$ and $P_{init} \implies \mathsf{stack}(\varepsilon)$.

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:

$$\{\mathsf{stack_w}(\alpha) * I(\varepsilon) * \forall\beta, a.\, \mathsf{spec}(\{I(\beta)\}\ f(a)\ \{I(a :: \beta)\})\}$$
$$\pi_3(\mathbf{wrap}_{\mathsf{stack}}(ops))(f)\{\mathsf{stack_w}(\alpha) * I(rev(\alpha))\}$$

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 β so far.

$$I_{\mathsf{w}}(\beta) \stackrel{\text{def}}{=} I(\beta) * \exists t_1, t_2.\, \mathsf{stk_{tr}}(t_1, \alpha) \wedge \mathsf{trav}(t_2, rev(\beta), f) \wedge \mathsf{trace}(t_1 \cdot \langle \mathtt{call}, \mathtt{foreach}, f\rangle \cdot t_2)$$

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

[Context α, β, f : Val]
$\{I_{\mathsf{w}}(\beta)\}$
$\{I(\beta) * \exists t_1, t_2.\, \mathsf{stk_{tr}}(t_1, \alpha) * \mathsf{trav}(t_2, rev(\beta), f) * \mathsf{trace}(t_1 \cdot \langle \mathtt{call}, \mathtt{foreach}, f\rangle \cdot t_2)\}$
emit $\langle$call, f, $a\rangle$;
$\{I(\beta) * \exists t_1, t_2.\, \mathsf{stk_{tr}}(t_1, \alpha) * \mathsf{trav}(t_2, rev(\beta), f)$
$\quad * \mathsf{trace}(t_1 \cdot \langle \mathtt{call}, \mathtt{foreach}, f\rangle \cdot t_2 \cdot \langle \mathtt{call}, f, a\rangle)\}$
f(a);
$\{I(a :: \beta) * \exists t_1, t_2.\, \mathsf{stk_{tr}}(t_1, \alpha) * \mathsf{trav}(t_2, rev(\beta), f)$
$\quad * \mathsf{trace}(t_1 \cdot \langle \mathtt{call}, \mathtt{foreach}, f\rangle \cdot t_2 \cdot \langle \mathtt{call}, f, a\rangle)\}$
emit $\langle$ret, $f\rangle$;
$\{I(a :: \beta) * \exists t_1, t_2.\, \mathsf{stk_{tr}}(t_1, \alpha) * \mathsf{trav}(t_2, rev(\beta), f)$
$\quad * \mathsf{trace}(t_1 \cdot \langle \mathtt{call}, \mathtt{foreach}, f\rangle \cdot t_2 \cdot \langle \mathtt{call}, f, a\rangle \cdot \langle \mathtt{ret}, f\rangle)\}$
$\{I(a :: \beta) * \exists t_1, t_2.\, \mathsf{stk_{tr}}(t_1, \alpha) * \mathsf{trav}(t_2, rev(a :: \beta), f) * \mathsf{trace}(t_1 \cdot \langle \mathtt{call}, \mathtt{foreach}, f\rangle \cdot t_2)\}$
$\{I_{\mathsf{w}}(a :: \beta)\}$

The second to last step follows from the following property:

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

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 Semantics

We give a denotational semantics for the logic introduced previously and establish soundness of the logic. The semantics is based on an interpretation of resources as members of a suitable resource monoid M. In fact, M is a partial commutative monoid $(|\mathsf{M}|, \bullet, 1)$ whereby the multiplication operator acts as the semantic analogue of separating conjunction. We shall also use the partial order relation yielded by monoid multiplication: $m_1 \leq m_2 \iff \exists m \in |\mathsf{M}|.\, m_1 \bullet m = m_2$ for all $m_1, m_2 \in |\mathsf{M}|$. In the sequel we shall frequently abuse notation and write $|\mathsf{M}|$ simply as M. The semantics of resources is parametrised on *worlds*, that is, partially ordered sets W specifying functional environments and trace invariants.

The monoid models heap resources and trace resources. It is constructed as the product of two partial commutative monoids: $\mathsf{M} \stackrel{\text{def}}{=} \mathit{Heap} \times \mathit{Trace}$, where:

$$\mathit{Heap} \stackrel{\text{def}}{=} (\mathit{Heap}, \uplus, \emptyset) \qquad \mathit{Trace} \stackrel{\text{def}}{=} (\{\mathsf{hs}, \mathsf{tr}\} \times \mathit{Val}^*, \bullet, (\mathsf{hs}, \varepsilon))$$

The monoid multiplication on *Heap* is disjoint union ($\uplus$), which is only defined between heaps with disjoint domains. The empty heap $\emptyset$ is the unit.

The monoid multiplication for the trace monoid is defined by:

$$(\mathsf{hs}, t_1) \bullet (\mathsf{hs}, t_2) \stackrel{\text{def}}{=} \begin{cases} (\mathsf{hs}, t_1) & \text{if } t_2 \leq_{pref} t_1 \\ (\mathsf{hs}, t_2) & \text{if } t_1 \leq_{pref} t_2 \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$(\mathsf{tr}, t_1) \bullet (\mathsf{hs}, t_2) \stackrel{\text{def}}{=} (\mathsf{hs}, t_2) \bullet (\mathsf{tr}, t_1) \stackrel{\text{def}}{=} \begin{cases} (\mathsf{tr}, t_1) & \text{if } t_2 \leq_{pref} t_1 \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$(\mathsf{tr}, t_1) \bullet (\mathsf{tr}, t_2) \quad \text{undefined}$$

where $\leq_{pref}$ is prefix ordering on finite sequences. The idea is that $\mathsf{trace}(t)$ is modelled by (tr, t), while $\mathsf{hist}(t)$ is modelled by (hs, t). The monoid multiplication ensures that (tr, t) is a unique resource, which grants the right to extend the trace. (It does not permit arbitrary changes to the trace: updates must preserve all frames, and hence all prefixes of the trace.) The resource (hs, t) is duplicable, and only ensures that t is a prefix of the trace. It is easy to see that $(\mathsf{hs}, \varepsilon)$ (where ε denotes the empty sequence) is the unit of this monoid.

Worlds model the information contained in assertions that does not behave like a resource. Worlds include the function environment and a trace invariant. We thus define $\mathsf{W} \stackrel{\text{def}}{=} \mathit{FEnv} \times \mathcal{P}(\mathit{Val}^*)$ with $(\gamma_1, I_1) \leq (\gamma_2, I_2)$ iff $\gamma_1 \subseteq \gamma_2$ and $I_1 = I_2$. The ordering on worlds describes how they may change with time. This ordering allows new functions to be named, but enforces that the trace invariant does not change with time.

Assertions are interpreted as monotone functions from worlds to upwards closed sets of resources: $[\![\mathsf{Prop}]\!] \stackrel{\text{def}}{=} \mathsf{W} \to_{\text{mon}} \mathcal{P}^{\uparrow}(\mathsf{M})$ where $\mathcal{P}^{\uparrow}(\mathsf{M}) \stackrel{\text{def}}{=} \{ p \subseteq \mathsf{M} \mid \forall m \in p. \forall m' \in \mathsf{M}.\ m \leq m' \implies m' \in p \}$. The rest of the types are interpreted as shown below, where the ordering on $\{\bot, \top\}$ is $\bot < \top$.

$$[\![1]\!] \stackrel{\text{def}}{=} \{*\} \quad [\![\mathsf{Bool}]\!] \stackrel{\text{def}}{=} \{\mathsf{true}, \mathsf{false}\} \quad [\![\mathsf{Loc}]\!] \stackrel{\text{def}}{=} \mathit{Loc} \quad [\![\tau \to \sigma]\!] \stackrel{\text{def}}{=} [\![\tau]\!] \to [\![\sigma]\!]$$

$$[\![\mathsf{Nat}]\!] \stackrel{\text{def}}{=} \mathbb{N} \quad [\![\mathsf{Val}]\!] \stackrel{\text{def}}{=} \mathit{Val} \quad [\![\mathsf{Exp}]\!] \stackrel{\text{def}}{=} \mathit{Exp} \quad [\![\tau \times \sigma]\!] \stackrel{\text{def}}{=} [\![\tau]\!] \times [\![\sigma]\!]$$

$$[\![\mathsf{seq}\,\tau]\!] \stackrel{\text{def}}{=} ([\![\tau]\!])^* \qquad [\![\mathsf{Spec}]\!] \stackrel{\text{def}}{=} \mathsf{W} \to_{\text{mon}} \{\bot, \top\}$$

The semantics of a term $\Gamma \vdash M : \tau$ is defined inductively as in Figure 3. (We give selected cases; for full details see Appendix B.) The semantics is defined in terms of a variable environment $\rho \in [\![\Gamma]\!]$ that maps variables of type τ to elements of $[\![\tau]\!]$: $[\![\Gamma]\!] = \{\rho : \mathsf{dom}(\Gamma) \to \bigcup_\tau [\![\tau]\!] \mid \forall (x : \tau) \in \Gamma.\ \rho(x) \in [\![\tau]\!]\}$

The definition of the weakest precondition $\mathsf{wp}(Q)$, used to define the semantics of Hoare triples, enforces the invariant on traces generated by the considered

$$
\begin{aligned}
[\![\Gamma \vdash P \Rightarrow Q : \mathsf{Prop}]\!]_\rho(w) &\stackrel{\text{def}}{=} \left\{ m \;\middle|\; \begin{array}{l} \forall m' \geq m. \forall w' \geq w.\, m' \in [\![\Gamma \vdash P : \mathsf{Prop}]\!]_\rho(w') \\ \implies m' \in [\![\Gamma \vdash Q : \mathsf{Prop}]\!]_\rho(w') \end{array} \right\} \\
[\![\Gamma \vdash P * Q : \mathsf{Prop}]\!]_\rho(w) &\stackrel{\text{def}}{=} \left\{ m \;\middle|\; \begin{array}{l} \exists m_1, m_2.\, m_1 \in [\![\Gamma \vdash P : \mathsf{Prop}]\!]_\rho(w) \\ \wedge m_2 \in [\![\Gamma \vdash Q : \mathsf{Prop}]\!]_\rho(w) \wedge m = m_1 \bullet m_2 \end{array} \right\} \\
[\![\Gamma \vdash M \mapsto N : \mathsf{Prop}]\!]_\rho(w) &\stackrel{\text{def}}{=} \{ m \in \mathsf{M} \mid m([\![\Gamma \vdash M : \mathsf{Loc}]\!]_\rho) = [\![\Gamma \vdash N : \mathsf{Val}]\!]_\rho \} \\
[\![\Gamma \vdash \mathsf{spec}(M) : \mathsf{Prop}]\!]_\rho(w) &\stackrel{\text{def}}{=} \{ m \in \mathsf{M} \mid [\![\Gamma \vdash M : \mathsf{Spec}]\!]_\rho(w) = \top \} \\
[\![\Gamma \vdash \mathsf{valid}(P) : \mathsf{Spec}]\!]_\rho(w) &\stackrel{\text{def}}{=} (\, [\![\Gamma \vdash P : \mathsf{Prop}]\!]_\rho(w) = \mathsf{M} \,) \\
[\![\Gamma \vdash \mathsf{trace}(t) : \mathsf{Prop}]\!]_\rho(w) &\stackrel{\text{def}}{=} \{ (h, \tau) \mid \tau \geq (\mathsf{tr}, [\![\Gamma \vdash t : \mathsf{seq\ Val}]\!]_\rho) \} \\
[\![\Gamma \vdash \mathsf{hist}(t) : \mathsf{Prop}]\!]_\rho(w) &\stackrel{\text{def}}{=} \{ (h, \tau) \mid \tau \geq (\mathsf{hs}, [\![\Gamma \vdash t : \mathsf{seq\ Val}]\!]_\rho) \} \\
[\![\Gamma \vdash \mathsf{inv}(I) : \mathsf{Prop}]\!]_\rho(w) &\stackrel{\text{def}}{=} \{ m \mid \pi_2(w) = \{ t \mid [\![\Gamma \vdash I : \mathsf{seq\ Val} \to \mathsf{Bool}]\!]_\rho(t) = \mathsf{true} \} \} \\
[\![\Gamma \vdash \{P\}\ e\ \{Q\} : \mathsf{Spec}]\!]_\rho(w) &\stackrel{\text{def}}{=} \forall w' \geq w.\, [\![\Gamma \vdash P : \mathsf{Prop}]\!]_\rho(w') \\
&\qquad\qquad \subseteq \mathsf{wp}([\![\Gamma \vdash Q : \mathsf{Val} \to \mathsf{Prop}]\!]_\rho)(\rho(e))(w') \\
[\![\Gamma \vdash M =_\tau N : \mathsf{Spec}]\!]_\rho(w) &\stackrel{\text{def}}{=} (\, [\![\Gamma \vdash M : \tau]\!]_\rho = [\![\Gamma \vdash N : \tau]\!]_\rho \,)
\end{aligned}
$$

$$
\mathsf{wp}(Q) \stackrel{\text{def}}{=} \nu wp'.\, \lambda e, w. \left\{ m \;\middle|\; \begin{array}{l} \forall r, t, s.\ t, s \vDash_w m \bullet r \implies \\ \quad (e, s \not\to \implies e \in \mathit{Val} \wedge m \in Q(e)(w)) \\ \quad \wedge \forall a, e', s'.\ e, s \xrightarrow{a} e', s' \implies \\ \qquad \exists w' \geq w, m'.\, (t \cdot a), s' \vDash_{w'} m' \bullet r \wedge m' \in wp'(e')(w') \end{array} \right\}
$$

$$
\begin{aligned}
t, (h, \gamma) \vDash_{(\gamma', I)} (h', (\mathsf{tr}, t')) &\stackrel{\text{def}}{=} (t = t' \wedge h = h' \wedge \gamma = \gamma' \wedge t \in I) \\
t, (h, \gamma) \vDash_{(\gamma', I)} (h', (\mathsf{hs}, t')) &\stackrel{\text{def}}{=} (t' \leq_{\mathit{pref}} t \wedge h = h' \wedge \gamma = \gamma' \wedge t \in I)
\end{aligned}
$$

Fig. 3. Semantics of terms (selected cases).

terms. It is defined as a greatest fixed-point, which establishes that the updates to the concrete state (the trace t, heap h, and function context γ) simulate updates to the abstract state (the resource m and world w), with respect to the erasure relation ($\vDash$). Updates to the concrete state are according to the operational semantics, while updates to the abstract state must preserve frames (r) and increase the world. When a terminal configuration is reached, the abstract state must satisfy the postcondition.

The semantics of entailment in the logic is defined as:

$$
\begin{aligned}
\Gamma \mid \Theta \models S &\stackrel{\text{def}}{=} \forall w \in \mathsf{W}. \forall \rho \in [\![\Gamma]\!].\ [\![\Gamma \vdash \Theta]\!]_\rho(w) \leq [\![\Gamma \vdash S : \mathsf{Spec}]\!]_\rho(w) \\
\Gamma \mid \Theta \mid P \models Q &\stackrel{\text{def}}{=} \forall w \in \mathsf{W}. \forall \rho \in [\![\Gamma]\!].\ [\![\Gamma \vdash \Theta]\!]_\rho(w) = \top \\
&\qquad \implies [\![\Gamma \vdash P : \mathsf{Prop}]\!]_\rho(w) \subseteq [\![\Gamma \vdash Q : \mathsf{Prop}]\!]_\rho(w)
\end{aligned}
$$

where $[\![\Gamma \vdash \Theta]\!]_\rho(w) = \bigwedge_{T \in \Theta} [\![\Gamma \vdash T : \mathsf{Spec}]\!]_\rho(w)$ (here $\bigwedge$ is lub in $\{\bot, \top\}$).

Soundness of the logic is proved by induction on the structure of derivations (cf. Appendix B). Using soundness, we can relate a proof of a triple $\{P\}\ e\ \{Q\}$ to the store and trace obtained from the reduction of e.

Theorem 3 (Soundness). *If $\Gamma \mid \Theta \vdash S$ then $\Gamma \mid \Theta \models S$.*

Corollary 1. *Suppose $\Gamma \mid - \vdash \{P\}\ e\ \{Q\}$ and let $w \in \mathsf{W}, \rho \in [\![\Gamma]\!]$ and $m \in [\![\Gamma \vdash P]\!]_\rho(w)$. Then, for all r, t, s such that $t, s \vDash_w m$ and $\rho(e), s \xrightarrow{t'}{}^{*} e', s' \not\to$, for some s', t', we have that $e' \in \mathit{Val}$ and $\exists w' \geq w. \exists m' \in [\![\Gamma \vdash Q]\!]_\rho(e')(w'). (t \cdot t'), s' \vDash_{w'} m'$.*

7 Conclusions

In this paper we demonstrated a formal approach for relating library specifications, expressed in separation logic, with the trace properties they enforce on the interaction between clients verified against the specified library and the library itself. The distinctive strength of our technique is that it is based purely on the abstract library specification and is independent of both client and library implementations. As such, it differs from the standard verification approach where one verifies a program against a given specification expressing the desired property. Since our main goal has been to establish a theoretical foundation relating specifications and trace properties, we focused on expressiveness rather than automation.

7.1 Related Work

Several lines of work have targeted static verification of safety trace properties of object-oriented and higher-order programs.

A particularly influential approach has been Typestates [29,8]. 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 [3,4] 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 [22].

Static analyses based on type systems have been widely used to check resource usage, like our file module example. Linear types are used in [7] 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 [14].

Type and effect systems have been used in [28] 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 [27], 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 μ-calculus [18].

Those approaches have been designed to support automated static verification and thus trade off expressiveness for automation. Here 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 [1]), others that rely on tracking an unbounded number of objects (5.2 and Appendix A, where we need to track all valid iterators and strings respectively), and we can even go beyond context-free languages (5.4, the language requires an order-2 pushdown automaton [21]). On an orthogonal direction, working with a logic with quantification we can specify traces from infinite alphabets of trace events (5.2, 5.3, 5.4 and Appendix A).

Our key contribution is a technique for formally relating separation logic specifications with the temporal properties they enforce, through wrapping abstract resources with assertions about traces. On the other hand, a very active line of work has targeted the verification of fine-grained concurrent data structures using program logics with histories [11,9] and separation logics [26,25]. Fine-grained concurrent algorithms use locking at the level of individual memory operations and their correctness often relies on subtle temporal properties about internal interactions within libraries. These approaches include primitives for reasoning explicitly about traces [11,9] or assign to (fine-grained) separation logic primitives history-oriented interpretations [26]. Temporal reasoning is thus achieved in a different way than herein, namely by specifying and verifying the underlying libraries themselves, whereas we derive temporal properties from specifications that are not themselves temporal.

The F7 and F^* programming languages provide another technique for reasoning about trace properties [30,2]. 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 [6]. 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 comparison to our work, the 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.

7.2 Further Directions

We demonstrated the power of even a basic higher-order separation logic for enforcing elaborate trace properties. A more expressive logic with better support for shared-resource reasoning would allow us to enforce even more elaborate protocols. In the future we therefore intend to apply our technique to a fully featured concurrent higher-order separation logic [17].

In the concurrent setting, linearisability [13] is a trace property that is commonly used to specify that operations of a library behave as if they were atomic. Several concurrent separation logics adopt a different approach to specifying atomicity, due to Jacobs and Piessens [16]. Although there is a strong intuitive argument that this approach implies linearisability, a formal connection has not been made. Our technique could be used to formalise such a connection.

Another direction concerns the description of the captured protocols via formal language tools, such as automata. Note that such a step is not to be taken lightly, as one would need to establish a formal link between the description and the logical specification. A related area for future work would be a formal (and, ideally, automated) procedure for deriving separation logic specifications for enforcing a given protocol from a formal definition of the protocol.

References

1. Alur, R., Madhusudan, P.: Visibly pushdown languages. In: Proceedings of the 36th Annual ACM Symposium on Theory of Computing, Chicago, IL, USA, June 13-16, 2004. pp. 202–211 (2004)
2. Bengtson, J., Bhargavan, K., Fournet, C., Gordon, A.D., Maffeis, S.: Refinement Types for Secure Implementations. ACM Trans. Program. Lang. Syst. 33(2), 8:1–8:45 (Feb 2011)
3. Bierhoff, K., Aldrich, J.: Modular typestate checking of aliased objects. In: Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications. pp. 301–320. OOPSLA'07, ACM (2007)
4. Bierhoff, K., Beckman, N.E., Aldrich, J.: Practical API protocol checking with access permissions. In: Proceedings of the 23rd European Conference on Object-Oriented Programming. pp. 195–219. ECOOP'09, Springer-Verlag (2009)
5. Biering, B., Birkedal, L., Torp-Smith, N.: Bi-hyperdoctrines, higher-order separation logic, and abstraction. ACM Trans. Program. Lang. Syst. 29(5) (Aug 2007)
6. Borgström, J., Gordon, A.D., Pucella, R.: Roles, Stacks, Histories: A Triple for Hoare. Journal of Functional Programming 21, 159–207 (2011)
7. DeLine, R., Fähndrich, M.: Enforcing high-level protocols in low-level software. In: Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation. pp. 59–69. PLDI '01, ACM (2001)
8. DeLine, R., Fähndrich, M.: Typestates for objects. In: Object-Oriented Programming: Proceedings of the 18th European Conference on Object-Oriented Programming. p. 465. ECOOP'04, Springer-Verlag (2004)
9. Fu, M., Li, Y., Feng, X., Shao, Z., Zhang, Y.: Reasoning about Optimistic Concurrency Using a Program Logic for History. In: In Proceedings of CONCUR (2010)
10. Gardner, P., Ntzik, G., Wright, A.: Local reasoning for the POSIX file system. In: ESOP. pp. 169–188 (2014)
11. Gotsman, A., Rinetzky, N., Yang, H.: Verifying concurrent memory reclamation algorithms with grace. In: Proceedings of ESOP (2013)
12. Grigore, R., Distefano, D., Petersen, R.L., Tzevelekos, N.: Runtime verification based on register automata. In: Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 260–276. TACAS'13, Springer-Verlag (2013)
13. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. 12(3), 463–492 (Jul 1990)
14. Igarashi, A., Kobayashi, N.: Resource usage analysis. In: Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 331–342. POPL '02, ACM (2002)
15. Ishtiaq, S.S., O'Hearn, P.W.: Bi as an assertion language for mutable data structures. In: Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 14–26. POPL'01, ACM (2001)
16. Jacobs, B., Piessens, F.: Expressive modular fine-grained concurrency specification. In: POPL. pp. 271–282 (2011)
17. Jung, R., Swasey, D., Sieczkowski, F., Svendsen, K., Turon, A., Birkedal, L., Dreyer, D.: Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In: Proceedings of POPL (2015)
18. Kobayashi, N.: 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. pp. 416–428. POPL '09, ACM (2009)

19. Krishnaswami, N.R., Aldrich, J., Birkedal, L., Svendsen, K., Buisse, A.: Design patterns in separation logic. In: Proceedings of the 4th International Workshop on Types in Language Design and Implementation. pp. 105–116. TLDI'09, ACM (2009)
20. Laird, J.: A fully abstract trace semantics for general references. In: Proceedings of the 34th International Conference on Automata, Languages and Programming. pp. 667–679. ICALP'07, Springer-Verlag (2007)
21. Maslov, A.N.: Multilevel stack automata. Problems Inform. Transmission 12(1), 38–42 (1976)
22. Naeem, N.A., Lhoták, O.: Extending typestate analysis to multiple interacting objects. OOPSLA08: Proceedings of Object-Oriented Programming, Systems, Languages and Applications (2008)
23. Parkinson, M., Bierman, G.: Separation logic and abstraction. In: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 247–258. POPL'05, ACM (2005)
24. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science. pp. 55–74. LICS'02, IEEE (2002)
25. da Rocha Pinto, P., Dinsdale-Young, T., Gardner, P.: TaDA: A Logic for Time and Data Abstraction. In: Proceedings of ECOOP (2014)
26. Sergey, I., Nanevski, A., Banerjee, A.: Specifying and Verifying Concurrent Algorithms with Histories and Subjectivity. In: Proceedings of ESOP (2015)
27. Skalka, C.: Types and trace effects for object orientation. Higher-Order and Symbolic Computation 21(3), 239–282 (2008)
28. Skalka, C., Smith, S., Van Horn, D.: Types and trace effects of higher order programs. Journal of Functional Programming 18(02), 179–249 (2008)
29. Strom, R.E., Yemini, S.: Typestate: A programming language concept for enhancing software reliability. IEEE Trans. Softw. Eng. 12(1), 157–171 (Jan 1986)
30. Swamy, N., Chen, J., Fournet, C., Strub, P.Y., Bhargavan, K., Yang, J.: Secure Distributed Programming with Value-Dependent Types. Journal of Functional Programming 23(4), 402–451 (2013)

A String Sanitisation

Another interesting example is string sanitisation [12]. Suppose strings are fetched from a web form, processed internally and passed as part of an SQL query to a database server. To avoid injection attacks, all inputs from the web form should be sanitized before being passed to the database server. We can express this as a *taint protocol*. Strings received from the user are considered *tainted* initially. Taint remains with a string and is also passed to any string produced by processing some tainted string. The only way to remove taint from a string is by *sanitising* it. A protocol we may require is:

No tainted string can reach the database server.

To express this trace property formally, we first define a suitable wrapping around a string library, that emits events describing the interaction between library and client. Suppose that a string library consists of five methods: an `input` method for obtaining strings from the user via a web form, a `constant` method for declaring string constants in the code, a `sanitize` method, a concatenation method, `concat`, and a sink method, `sink`, for sending a given string to the database server. Formally, we represent the library as a 5-tuple of these operations. We can now define a wrapper that takes a string library and returns an instrumented string library. 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:

$$\begin{aligned}
&\mathbf{wrap}_{\mathsf{str}} \stackrel{\text{def}}{=} \lambda(\mathit{input}, \mathit{constant}, \mathit{sanitize}, \mathit{concat}, \mathit{sink}).\ (\\
&\quad \lambda_.\ \mathsf{let}\ r = \mathit{input}()\ \mathsf{in}\ \mathsf{emit}\langle \mathtt{input}, r\rangle;\ r,\\
&\quad \lambda y.\ \mathsf{let}\ r = \mathit{constant}(y)\ \mathsf{in}\ \mathsf{emit}\langle \mathtt{constant}, r\rangle;\ r,\\
&\quad \lambda y.\ \mathit{sanitize}(y);\ \mathsf{emit}\langle \mathtt{sanitize}, y\rangle,\\
&\quad \lambda(y_1, y_2).\ \mathsf{let}\ r = \mathit{concat}(y_1, y_2)\ \mathsf{in}\ \mathsf{emit}\langle \mathtt{concat}, r, y_1, y_2\rangle;\ r,\\
&\quad \lambda y.\ \mathit{sink}(y);\ \mathsf{emit}\langle \mathtt{sink}, y\rangle\ )
\end{aligned}$$

We can now define the trace property as a constraint on the traces generated by linking a client with an instrumented library. We let the alphabet be the following countable set:

$$\begin{aligned}
\Sigma = &\{\langle e, s\rangle \mid e \in \{\mathtt{constant}, \mathtt{input}, \mathtt{sanitize}, \mathtt{sink}\}, s \in \mathit{Loc}\}\\
&\cup \{\langle \mathtt{concat}, s, s_1, s_2\rangle \mid s, s_1, s_2 \in \mathit{Loc}\}
\end{aligned}$$

Intuitively, we want to constrain traces such that for any $\langle \mathtt{sink}, s\rangle$ event in the trace, all the strings s' used to construct s are safe to emit, meaning that the trace contains a corresponding `constant` event or `sanitize` event for such s'. We first define, by induction on t, a predicate $\mathsf{esafe}(s, t)$ to express that the string s is safe to emit given that the current trace is t. We let $\mathsf{esafe}(s, \varepsilon) \stackrel{\text{def}}{=} \bot$ and:

$$\begin{aligned}
\mathsf{esafe}(s, h :: t) \stackrel{\text{def}}{=}\ &\mathsf{esafe}(s, t) \lor h = \langle \mathtt{constant}, s\rangle \lor h = \langle \mathtt{sanitize}, s\rangle \lor\\
&(\exists s_1, s_2.\ h = \langle \mathtt{concat}, s, s_1, s_2\rangle \land \mathsf{esafe}(s_1, t) \land \mathsf{esafe}(s_2, t))
\end{aligned}$$

Note such inductive predicates are definable in higher-order logic.

We wish to ensure that once a string has been sanitised it can never become tainted again or, stated in terms of esafe, once a string is safe for t then it is also safe for all future histories t' such that $t \leq_{pref} t'$. To ensure this we require the library to ensure that `input`, `constant` and `concat` always return fresh string pointers that have never been used before. We thus define the language of valid traces, $\mathcal{L}_{\mathsf{str}}$, as the set of all strings $t \in \Sigma^*$ such that $\mathsf{str}_{\mathsf{trace}}(t)$, where:

$$\begin{aligned}
\mathsf{str}_{\mathsf{trace}}(t) &\stackrel{\text{def}}{=} (\forall n, s.\, t[n] = \langle \mathtt{sink}, s\rangle \implies \mathsf{esafe}(s,t)) \vee \mathsf{notfresh}(t)\\
\mathsf{allocs}(s,t,n) &\stackrel{\text{def}}{=} t[n] = \langle \mathtt{constant}, s\rangle \vee t[n] = \langle \mathtt{input}, s\rangle \vee\\
&\qquad \exists s_1, s_2.\, t[n] = \langle \mathtt{concat}, s, s_1, s_2\rangle\\
\mathsf{notfresh}(t) &\stackrel{\text{def}}{=} \exists n, m, s.\, n \neq m \wedge \mathsf{allocs}(s,t,n) \wedge \mathsf{allocs}(s,t,m)
\end{aligned}$$

The $\mathsf{str}_{\mathsf{trace}}(t)$ predicate expresses that either every sink event in t uses a safe string s or the library does not ensure sufficient freshness of string pointers. In the latter case we do not constrain the client at all. We thus have the following crucial property, which ensures that if a string s is safe for the current trace t then it is also safe for any future trace t' with sufficiently fresh string pointers:

$$\forall s, t_1, t_2.\, t_1 \leq_{pref} t_2 \wedge \mathsf{esafe}(s,t_1) \implies \mathsf{esafe}(s,t_2) \vee \mathsf{notfresh}(t_2)$$

Below we define a separation logic specification for the string sanitisation library that ensures clients only call `sink` with safe strings. The specification uses a resource R for the local state of the sanitisation library, a string resource $\mathsf{str}(x)$ for each string and a safe resource $\mathsf{safe}(x)$ which expresses that the string x is safe. As expected, `sanitize` turns a string into a safe string and `concat` applied to safe strings yields a safe string.

$$\begin{aligned}
&\Phi_{\mathsf{str}}(P_0, (\mathit{input}, \mathit{constant}, \mathit{sanitize}, \mathit{concat}, \mathit{sink})) \stackrel{\text{def}}{=}\\
&\quad \exists \mathsf{R} : \mathsf{Prop}.\, \exists \mathsf{str}, \mathsf{safe} : \mathsf{Val} \to \mathsf{Prop}.\\
&\qquad \mathsf{valid}(P_0 \implies \mathsf{R}) \wedge \mathsf{valid}(\forall s.\, \mathsf{safe}(s) \implies \mathsf{safe}(s) * \mathsf{safe}(s)) \wedge\\
&\qquad \left\{\mathsf{R}\right\}\ \mathit{input}()\ \left\{r.\, \mathsf{R} * \mathsf{str}(r)\right\}\ \wedge \forall s.\, \left\{\mathsf{R}\right\}\ \mathit{constant}(s)\ \left\{r.\, \mathsf{R} * \mathsf{str}(r) * \mathsf{safe}(r)\right\}\ \wedge\\
&\qquad \forall s.\, \left\{\mathsf{R} * \mathsf{str}(r)\right\}\ \mathit{sanitize}(s)\ \left\{\mathsf{R} * \mathsf{str}(r) * \mathsf{safe}(s)\right\}\ \wedge\\
&\qquad \forall s_1, s_2.\, \{\mathsf{R} * \mathsf{str}(s_1) * \mathsf{str}(s_2)\}\quad \mathit{concat}(s_1, s_2)\{r.\, \mathsf{R} * \mathsf{str}(s_1) * \mathsf{str}(s_2) * \mathsf{str}(r)\} \wedge\\
&\qquad \forall s_1, s_2.\, \{\mathsf{R} * \mathsf{str}(s_1) * \mathsf{str}(s_2) * \mathsf{safe}(s_1) * \mathsf{safe}(s_2)\}\\
&\qquad\qquad \mathit{concat}(s_1, s_2)\{x.\, \mathsf{R} * \mathsf{str}(s_1) * \mathsf{str}(s_2) * \mathsf{str}(x) * \mathsf{safe}(x)\} \wedge\\
&\qquad \forall s.\, \left\{\mathsf{R} * \mathsf{str}(s) * \mathsf{safe}(s)\right\}\ \mathit{sink}(s)\ \left\{\mathsf{R} * \mathsf{str}(s)\right\}
\end{aligned}$$

As usual, we proceed by proving that the instrumentation preserves the library specification and generates traces in $\mathcal{L}_{\mathsf{str}}$.

Lemma 5. $P_0, \mathit{ops} \mid \Phi_{\mathit{str}}(P_0, \mathit{ops}) \vdash \Phi_{\mathit{str}}(P_0 * \mathsf{trace}(\varepsilon) * \mathsf{inv}(\mathcal{L}_{\mathsf{str}}), \mathbf{wrap}_{\mathit{str}}(\mathit{ops}))$.

Proof (Proof sketch). To prove that the instrumentation preserves the library specification we first have to define instrumented versions of the abstract representation predicates. We let the local state resource R_{w} take ownership of the

trace resource and let $\mathsf{safe_w}$ assert the existence of some trace prefix in which the given string is safe or for which the library failed to ensure sufficiently fresh string pointers:

$$\mathsf{R_w} \stackrel{\text{def}}{=} \exists t \in \mathcal{L}_{\mathsf{str}}.\, \mathsf{R} * \mathsf{trace}(t) * \mathsf{inv}(\mathcal{L}_{\mathsf{str}}), \qquad \mathsf{str_w}(s) \stackrel{\text{def}}{=} \mathsf{str}(s)$$
$$\mathsf{safe_w}(s) \stackrel{\text{def}}{=} \exists t \in \mathcal{L}_{\mathsf{str}}.\, \mathsf{safe}(s) * \mathsf{hist}(t) * (\mathsf{esafe}(s,t) \vee \mathsf{notfresh}(t))$$

It remains to show that the instrumentation preserves the specifications of each of the library methods. The most interesting cases are `sink`, and `concat` when on two safe strings. For the latter we have:

[Context s_1, s_2 : Val]
$\{\mathsf{R_w} * \mathsf{str_w}(s_1) * \mathsf{str_w}(s_2) * \mathsf{safe_w}(s_1) * \mathsf{safe_w}(s_2)\}$
$\{\mathsf{R} * \mathsf{str}(s_1) * \mathsf{str}(s_2) * \mathsf{safe}(s_1) * \mathsf{safe}(s_2) * \exists t, t_1, t_2 \in \mathcal{L}_{\mathsf{str}}.\, \mathsf{trace}(t) * \mathsf{inv}(\mathcal{L}_{\mathsf{str}})$
$\quad * \mathsf{hist}(t_1) * \mathsf{hist}(t_2) * (\mathsf{esafe}(s_1,t_1) \vee \mathsf{notfresh}(t_1)) * (\mathsf{esafe}(s_2,t_2) \vee \mathsf{notfresh}(t_2))\}$
let $s = \mathsf{concat}(s_1,\ s_2)$ **in**
$\{\mathsf{R} * \mathsf{str}(s_1) * \mathsf{str}(s_2) * \mathsf{str}(s) * \mathsf{safe}(s) * \exists t, t_1, t_2 \in \mathcal{L}_{\mathsf{str}}.\, \mathsf{trace}(t) * \mathsf{inv}(\mathcal{L}_{\mathsf{str}})$
$\quad * \mathsf{hist}(t_1) * \mathsf{hist}(t_2) * (\mathsf{esafe}(s_1,t_1) \vee \mathsf{notfresh}(t_1)) * (\mathsf{esafe}(s_2,t_2) \vee \mathsf{notfresh}(t_2))\}$
emit$\langle$concat, s, s_1, $s_2\rangle$;
$\{\mathsf{R} * \mathsf{str}(s_1) * \mathsf{str}(s_2) * \mathsf{safe}(s) * \exists t, t' \in \mathcal{L}_{\mathsf{str}}.$
$\quad \mathsf{trace}(t) * \mathsf{inv}(\mathcal{L}_{\mathsf{str}}) * \mathsf{hist}(t') * (\mathsf{esafe}(s,t') \vee \mathsf{notfresh}(t'))\}$
$\{\mathsf{R_w} * \mathsf{str_w}(s_1) * \mathsf{str_w}(s_2) * \mathsf{str_w}(s) * \mathsf{safe_w}(s)\}$

here we use the following properties of esafe to verify the emit:

$$\forall s, s_1, s_2, t.\, \mathsf{esafe}(s_1,t) \wedge \mathsf{esafe}(s_2,t) \implies \mathsf{esafe}(s, t \cdot \langle \mathtt{concat}, s, s_1, s_2\rangle)$$
$$\forall s, s_1, s_2, t.\, \mathsf{esafe}(s_1,t) \wedge \mathsf{esafe}(s_2,t) \implies t \cdot \langle \mathtt{concat}, s, s_1, s_2\rangle \in \mathcal{L}_{\mathsf{str}}$$

For the sink case, we have:

[Context s : Val]
$\{\mathsf{R_w} * \mathsf{str_w}(s) * \mathsf{safe_w}(s)\}$
$\{\mathsf{R} * \mathsf{str}(s) * \mathsf{safe}(s) * \exists t, t' \in \mathcal{L}_{\mathsf{str}}.\, \mathsf{trace}(t) * \mathsf{inv}(\mathcal{L}_{\mathsf{str}}) * \mathsf{safe}(s) * \mathsf{hist}(t') * \mathsf{esafe}(s,t')\}$
$\mathsf{sink}(s)$;
$\{\mathsf{R} * \mathsf{str}(s) * \exists t, t' \in \mathcal{L}_{\mathsf{str}}.\, \mathsf{trace}(t) * \mathsf{inv}(\mathcal{L}_{\mathsf{str}}) * \mathsf{safe}(s) * \mathsf{hist}(t') * (\mathsf{esafe}(s,t') \vee \mathsf{notfresh}(t'))\}$
emit$\langle$sink, $s\rangle$;
$\{\mathsf{R} * \mathsf{str}(s) * \exists t \in \mathcal{L}_{\mathsf{str}}.\, \mathsf{trace}(t \cdot \langle \mathtt{sink}, s\rangle) * \mathsf{inv}(\mathcal{L}_{\mathsf{str}})\}$
$\{\mathsf{R_w} * \mathsf{str_w}(s)\}$

B Soundness

The semantics of all terms is given in Figure 4.

Theorem 3 (Soundess). *If* $\Gamma \mid \Theta \vdash S$ *then* $\Gamma \mid \Theta \models S$.

Soundness is established by showing that each of the proof rules is semantically valid.

As a notational convenience, given $f, g : \mathit{Exp} \to [\![\mathsf{Prop}]\!]$ we write $f \sqsubseteq g$ just if $\forall e, w.\, f(e)(w) \subseteq g(e)(w)$. We moreover define $f \sqcup g$ as $\lambda e, w.\, f(e)(w) \cup g(e)(w)$, and extend this notation to arbitrary unions $\bigsqcup S$ for $S \subseteq \mathit{Exp} \to [\![\mathsf{Prop}]\!]$.

$$\llbracket \Gamma \vdash x : \tau \rrbracket_\rho \stackrel{\text{def}}{=} \rho(x)$$

$$\llbracket \Gamma \vdash \bot : \mathsf{Prop} \rrbracket_\rho(w) \stackrel{\text{def}}{=} \emptyset$$

$$\llbracket \Gamma \vdash \top : \mathsf{Prop} \rrbracket_\rho(w) \stackrel{\text{def}}{=} \llbracket \Gamma \vdash \mathsf{emp} : \mathsf{Prop} \rrbracket_\rho(w) \stackrel{\text{def}}{=} \mathsf{M}$$

$$\llbracket \Gamma \vdash \lambda x : \tau.\, M : \tau \to \sigma \rrbracket_\rho \stackrel{\text{def}}{=} \lambda v \in \llbracket \tau \rrbracket.\, \llbracket \Gamma, x : \tau \vdash M : \sigma \rrbracket_{\rho[x \mapsto v]}$$

$$\llbracket \Gamma \vdash M\ N : \sigma \rrbracket_\rho \stackrel{\text{def}}{=} \llbracket \Gamma \vdash M : \tau \to \sigma \rrbracket_\rho(\llbracket \Gamma \vdash N : \tau \rrbracket_\rho)$$

$$\llbracket \Gamma \vdash \forall x : \tau.\, P : \mathsf{Prop} \rrbracket_\rho(w) \stackrel{\text{def}}{=} \bigcap_{v \in \llbracket \tau \rrbracket} \llbracket \Gamma, x : \tau \vdash P \rrbracket_{\rho[x \mapsto v]}(w)$$

$$\llbracket \Gamma \vdash P \wedge Q : \mathsf{Prop} \rrbracket_\rho(w) \stackrel{\text{def}}{=} \llbracket \Gamma \vdash P : \mathsf{Prop} \rrbracket_\rho(w) \cap \llbracket \Gamma \vdash Q : \mathsf{Prop} \rrbracket_\rho(w)$$

$$\llbracket \Gamma \vdash P \Rightarrow Q : \mathsf{Prop} \rrbracket_\rho(w) \stackrel{\text{def}}{=} \left\{ m \;\middle|\; \begin{array}{l} \forall m' \geq m.\, \forall w' \geq w.\, m' \in \llbracket \Gamma \vdash P : \mathsf{Prop} \rrbracket_\rho(w') \\ \qquad \implies m' \in \llbracket \Gamma \vdash Q : \mathsf{Prop} \rrbracket_\rho(w') \end{array} \right\}$$

$$\llbracket \Gamma \vdash P * Q : \mathsf{Prop} \rrbracket_\rho(w) \stackrel{\text{def}}{=} \left\{ m \;\middle|\; \begin{array}{l} \exists m_1, m_2.\, m_1 \in \llbracket \Gamma \vdash P : \mathsf{Prop} \rrbracket_\rho(w) \\ \wedge\, m_2 \in \llbracket \Gamma \vdash Q : \mathsf{Prop} \rrbracket_\rho(w) \wedge m = m_1 \bullet m_2 \end{array} \right\}$$

$$\llbracket \Gamma \vdash P \mathrel{-\!\!*} Q : \mathsf{Prop} \rrbracket_\rho(w) \stackrel{\text{def}}{=} \left\{ m \;\middle|\; \begin{array}{l} \forall w' \geq w.\, \forall m' \geq m.\, \forall m'' \in \mathsf{M}. \\ m' \bullet m'' \text{ defined} \wedge m'' \in \llbracket \Gamma \vdash P : \mathsf{Prop} \rrbracket_\rho(w') \\ \qquad \implies m' \bullet m'' \in \llbracket \Gamma \vdash Q : \mathsf{Prop} \rrbracket_\rho(w') \end{array} \right\}$$

$$\llbracket \Gamma \vdash M \mapsto N : \mathsf{Prop} \rrbracket_\rho(w) \stackrel{\text{def}}{=} \{\, m \in \mathsf{M} \mid m(\llbracket \Gamma \vdash M : \mathsf{Loc} \rrbracket_\rho) = \llbracket \Gamma \vdash N : \mathsf{Val} \rrbracket_\rho \}$$

$$\llbracket \Gamma \vdash \mathsf{spec}(M) : \mathsf{Prop} \rrbracket_\rho(w) \stackrel{\text{def}}{=} \{\, m \in \mathsf{M} \mid \llbracket \Gamma \vdash M : \mathsf{Spec} \rrbracket_\rho(w) = \top \,\}$$

$$\llbracket \Gamma \vdash M =_\tau N : \mathsf{Spec} \rrbracket_\rho(w) \stackrel{\text{def}}{=} (\, \llbracket \Gamma \vdash M : \tau \rrbracket_\rho = \llbracket \Gamma \vdash N : \tau \rrbracket_\rho \,)$$

$$\llbracket \Gamma \vdash \mathsf{valid}(P) : \mathsf{Spec} \rrbracket_\rho(w) \stackrel{\text{def}}{=} (\, \llbracket \Gamma \vdash P : \mathsf{Prop} \rrbracket_\rho(w) = \mathsf{M} \,)$$

$$\llbracket \Gamma \vdash \mathsf{trace}(t) : \mathsf{Prop} \rrbracket_\rho(w) \stackrel{\text{def}}{=} \{(h, \tau) \mid \tau \geq (\mathsf{tr}, \llbracket \Gamma \vdash t : \mathsf{seq\ Val} \rrbracket_\rho)\}$$

$$\llbracket \Gamma \vdash \mathsf{hist}(t) : \mathsf{Prop} \rrbracket_\rho(w) \stackrel{\text{def}}{=} \{(h, \tau) \mid \tau \geq (\mathsf{hs}, \llbracket \Gamma \vdash t : \mathsf{seq\ Val} \rrbracket_\rho)\}$$

$$\llbracket \Gamma \vdash \mathsf{inv}(I) : \mathsf{Prop} \rrbracket_\rho(w) \stackrel{\text{def}}{=} \{m \mid \pi_2(w) = \{t \mid \llbracket \Gamma \vdash I : \mathsf{seq\ Val} \to \mathsf{Bool} \rrbracket_\rho(t) = \mathsf{true}\}\}$$

$$\llbracket \Gamma \vdash \{P\}\ e\ \{Q\} : \mathsf{Spec} \rrbracket_\rho(w) \stackrel{\text{def}}{=} \begin{array}{l} \forall w' \geq w.\, \llbracket \Gamma \vdash P : \mathsf{Prop} \rrbracket_\rho(w') \\ \qquad \subseteq \mathsf{wp}(\llbracket \Gamma \vdash Q : \mathsf{Val} \to \mathsf{Prop} \rrbracket_\rho)(\rho(e))(w') \end{array}$$

$$\mathsf{wp}(Q) \stackrel{\text{def}}{=} \nu wp'.\, \lambda e, w. \left\{ m \;\middle|\; \begin{array}{l} \forall r, t, s.\;\; t, s \vDash_w m \bullet r \implies \\ \;(e, s \not\rightarrow \implies e \in \mathit{Val} \wedge m \in Q(e)(w)) \\ \;\wedge\, \forall a, e', s'.\;\; e, s \xrightarrow{a} e', s' \implies \\ \quad \exists w' \geq w, m'.\, (t \cdot a), s' \vDash_{w'} m' \bullet r \wedge m' \in wp'(e')(w') \end{array} \right\}$$

where
$$t, (h, \gamma) \vDash_{(\gamma', I)} (h', (\mathsf{tr}, t')) \iff t = t' \wedge h = h' \wedge \gamma = \gamma' \wedge t \in I$$
$$t, (h, \gamma) \vDash_{(\gamma', I)} (h', (\mathsf{hs}, t')) \iff t' \leq_{\mathit{pref}} t \wedge h = h' \wedge \gamma = \gamma' \wedge t \in I$$

Fig. 4. Semantics of terms (all cases).

Moreover, for each $p, q \in [\![\mathsf{Prop}]\!]$, we let $p{*}q$ stand for $\lambda w.\, p(w){*}q(w)$; whereby, for each $X, Y \subseteq \mathsf{M}$, $X{*}Y = \{m \in \mathsf{M} \mid \exists m_1, m_2.\, m = m_1{\bullet}m_2 \wedge m_1 \in X \wedge m_2 \in Y\}$.

Lemma 6 (Hyp rule).

$$\Gamma \mid \Theta, S \models S$$

Proof. Let $\rho \in [\![\Gamma]\!]$. Since $\forall w \in \mathsf{W}.\, \bigwedge_{T\in(\Theta,S)} [\![\Gamma \vdash T : \mathsf{Spec}]\!]_\rho(w) \leq [\![\Gamma \vdash S : \mathsf{Spec}]\!]_\rho(w)$, we have $\Gamma \mid \Theta, S \models S$ by definition.

Lemma 7 (Ret rule).

$$\Gamma, v : \mathsf{Val} \mid \Theta \models \{\top\}\ v\ \{r.\, r = v\}$$

Proof. Let $\rho \in [\![\Gamma, v : \mathsf{Val}]\!]$ and $w \in \mathsf{W} = \mathit{FEnv} \times \mathit{Trace}$.

$$\begin{aligned}
&\mathsf{wp}([\![\Gamma, v : \mathsf{Val} \vdash \lambda r.\, r = v : \mathsf{Val} \to \mathsf{Prop}]\!]_\rho)(\rho(v))(w)\\
&= \{m \in \mathsf{M} \mid m \in [\![\Gamma, v : \mathsf{Val} \vdash \lambda r.\, r = v : \mathsf{Val} \to \mathsf{Prop}]\!]_\rho(w)(\rho(v))\}\\
&= \mathsf{M}\\
&\supseteq [\![\Gamma, v : \mathsf{Val} \vdash \top : \mathsf{Prop}]\!]_\rho(w)
\end{aligned}$$

Hence $\Gamma \mid \Theta \models \{\top\}\ v\ \{r.\, r = v\}$.

Lemma 8 (Monotonicity of wp). *For all $p_1, p_2 : \mathit{Val} \to [\![\mathsf{Prop}]\!]$ and $w \in \mathsf{W}$, if $\forall v, w' \geq w.\, p_1(v)(w') \subseteq p_2(v)(w')$ then $\forall e, w' \geq w.\, \mathsf{wp}(p_1)(e)(w') \subseteq \mathsf{wp}(p_2)(e)(w')$.*

Proof. Straightforward, by co-induction.

Lemma 9. *For all $q : \mathit{Val} \to \mathit{Val} \to [\![\mathsf{Prop}]\!]$, $e \in \mathit{Exp}$, evaluation contexts K, and $w \in \mathsf{W}$,*

$$\mathsf{wp}(\lambda v.\, \mathsf{wp}(q(v))(K[v]))(e)(w) \subseteq \mathsf{wp}(\bigsqcup q)(K[e])(w)$$

where $\bigsqcup q \stackrel{def}{=} \lambda u.\, \lambda w.\, \cup_{v\in[\![\mathsf{Val}]\!]} q(v)(u)(w)$.

Proof. By co-induction. Assume $m \in \mathsf{wp}(\lambda v.\, \mathsf{wp}(q(v))(K[v]))(e)(w)$ and that r, t, s are such that $t, s \vDash_w m \bullet r$. If $K[e], s \not\to$ then $e, s \not\to$ and thus $e \in \mathit{Val}$ and $m \in \mathsf{wp}(q(e))(K[e])(w)$ as required, using continuity of wp. Suppose that $K[e], s \xrightarrow{a} e', s'$. If $e \in \mathit{Val}$ then, as above, $m \in \mathsf{wp}(q(e))(K[e])(w)$. Otherwise, there exists e'' such that $e' = K[e'']$ and $e, s \xrightarrow{a} e'', s'$. It then follows from the assumption that there exist $w' \geq w$ and m' with $(t \cdot a), s' \vDash_{w'} m' \bullet r$ and $m' \in \mathsf{wp}(\lambda v.\, \mathsf{wp}(q(v))(K[v]))(e'')(w')$. By the co-inductive assumption, $m' \in \mathsf{wp}(\bigsqcup q)(K[e''])(w')$ and so $m \in \mathsf{wp}(\bigsqcup q)(K[e])(w)$ as required.

Corollary 2 (Bind rule). *If*

$$\Gamma \mid \Theta \models \{P\}\ e\ \{x.\, Q\}$$
$$\Gamma, x : \mathsf{Val} \mid \Theta \models \{Q\}\ K[x]\ \{r.\, R\}$$

where $x \notin FV(\Theta)$, then

$$\Gamma \mid \Theta \models \{P\}\ K[e]\ \{r.\, \exists x : \mathsf{Val}.\, R\}$$

Proof. Follows from Lemmas 9 and 8.

Corollary 3 (Csq rule). *If*

$$\Gamma \mid \Theta \mid P_1 \models P_2$$
$$\Gamma \mid \Theta \models \{P_2\}\ e\ \{Q_2\}$$
$$\Gamma, x : \mathsf{Val} \mid \Theta \mid Q_2(x) \models Q_1(x)$$

then

$$\Gamma \mid \Theta \models \{P_1\}\ e\ \{Q_1\}$$

Proof. Let $\rho \in [\![\Gamma]\!]$ and $w \in \mathsf{W}$ be such that $[\![\Gamma \vdash \Theta]\!]_\rho(w)$. Suppose that $m \in [\![\Gamma \vdash P_1 : \mathsf{Prop}]\!]_\rho(w)$. By the first assumption, it follows that $m \in [\![\Gamma \vdash P_2 : \mathsf{Prop}]\!]_\rho(w)$. By the second assumption, it follows that $m \in \mathsf{wp}([\![\Gamma \vdash Q_2 : \mathsf{Val} \to \mathsf{Prop}]\!]_\rho)(\rho(e))(w)$. By the third assumption, we have $[\![\Gamma \vdash Q_2 : \mathsf{Val} \to \mathsf{Prop}]\!]_\rho \sqsubseteq [\![\Gamma \vdash Q_1 : \mathsf{Val} \to \mathsf{Prop}]\!]_\rho$. By Lemma 8, it then follows that $m \in \mathsf{wp}([\![\Gamma \vdash Q_1 : \mathsf{Val} \to \mathsf{Prop}]\!]_\rho)(\rho(e))(w)$, as required.

Lemma 10 (Framing). *For all* $p : [\![\mathsf{Prop}]\!]$, $q : [\![\mathsf{Val}]\!] \to [\![\mathsf{Prop}]\!]$, $e \in \mathit{Exp}$ *and* $w \in \mathsf{W}$,

$$(p * \mathsf{wp}(q)(e))(w) \subseteq \mathsf{wp}(\lambda v.\, p * q(v))(e)(w)$$

Proof. By co-induction. It suffices to show that, for all e, w, $(p * \mathsf{wp}(q)(e))(w) \subseteq H(\lambda e'.\, p * \mathsf{wp}(q)(e'))(e)(w)$ where H is such that $\mathsf{wp}(\lambda v.\, p * q(v)) = \nu wp'.\, H(wp')$ (as implied in the definition of wp). Take any w, $m \in (p * \mathsf{wp}(q)(e))(w)$, r, t, s with $s \vDash_w m \bullet r$. Then there exist m_1, m_2 such that $m' = m_1 \bullet m_2$, $m_1 \in p(w)$ and $m_2 \in \mathsf{wp}(q)(e)(w)$. If $e, s \not\to$ then, since $t, s \vDash_w m_2 \bullet (m_1 \bullet r)$, it follows that $e \in \mathit{Val}$ and $m_2 \in q(e)(w)$; we can hence conclude that $m \in H(\lambda e'.\, p * \mathsf{wp}(q)(e'))(e)(w)$. If $e, s \xrightarrow{a} e', s'$ then since $m_2 \in \mathsf{wp}(q)(e)(w)$, it follows that there are $w' \geq w$ and there exists m_2' such that $(t \cdot a), s' \vDash_{w'} m_2' \bullet (r \bullet m_1)$ and $m_2' \in \mathsf{wp}(q)(e')(w')$; hence $m_1 \bullet m_2' \in (p * \mathsf{wp}(q)(e'))(w')$ and we thus have that $m \in H(\lambda e'.\, p * \mathsf{wp}(q)(e'))(e)(w)$, as required.

Corollary 4 (Frame rule). *If* $\Gamma \mid \Theta \models \{P\}\ e\ \{Q\}$ *then* $\Gamma \mid \Theta \models \{P * R\}\ e\ \{\lambda r.\, Q(r) * R\}$.

Proof. Let $\rho \in [\![\Gamma]\!]$ and $w \in \mathsf{W}$ be such that $[\![\Gamma \vdash \Theta]\!]_\rho(w)$. By assumption,

$$[\![\Gamma \vdash P : \mathsf{Prop}]\!]_\rho(w) \subseteq \mathsf{wp}([\![\Gamma \vdash Q : \mathsf{Val} \to \mathsf{Prop}]\!]_\rho)(\rho(e))(w)$$

It follows by monotonicity of $*$ and Lemma 10 that

$$\begin{aligned}
&[\![\Gamma \vdash P * R : \mathsf{Prop}]\!]_\rho(w) \\
&= [\![\Gamma \vdash P : \mathsf{Prop}]\!]_\rho(w) * [\![\Gamma \vdash R : \mathsf{Prop}]\!]_\rho(w) \\
&\subseteq \mathsf{wp}([\![\Gamma \vdash Q : \mathsf{Val} \to \mathsf{Prop}]\!]_\rho)(\rho(e))(w) * [\![\Gamma \vdash R : \mathsf{Prop}]\!]_\rho(w) \\
&\subseteq \mathsf{wp}(\lambda v.\, [\![\Gamma \vdash Q : \mathsf{Val} \to \mathsf{Prop}]\!]_\rho(v) * [\![\Gamma \vdash R : \mathsf{Prop}]\!]_\rho)(\rho(e))(w)
\end{aligned}$$

as required.

Lemma 11 (Abs rule). *If*

$$\Gamma, x : \mathsf{Val} \mid \Theta \models \{P\}\ e\ \{Q\}$$

where $x \notin FV(\Theta)$, then

$$\Gamma \mid \Theta \models \{\mathsf{emp}\}\ \lambda x.\, e\ \{r.\, \forall x : \mathsf{Val}.\, \mathsf{spec}(\{P\}\ rx\ \{Q\})\}$$

Proof. Let $\rho \in [\![\Gamma]\!]$ and $w \in \mathsf{W}$ be such that $[\![\Gamma \vdash \Theta]\!]_\rho(w)$. Let $m \in \mathsf{M}$. Suppose that $t, (h, \gamma) \vDash_w m \bullet r$ and $(\lambda x.\, \rho(e)), (h, \gamma) \xrightarrow{a} f, (h', \gamma')$. Then it must be that $a = \epsilon$, $f \in Fun \backslash \mathsf{dom}(\gamma)$, $h' = h$ and $\gamma' = \gamma[f \mapsto \lambda x.\, \rho(e)]$. Let $w' = (\gamma', \pi_2(w)) = (\pi_1(w)[f \mapsto \lambda x.\, \rho(e)], \pi_2(w))$, so $w' \geq w$ and $t, (h, \gamma') \vDash_{w'} m \bullet r$. Moreover, $[\![\Gamma \vdash \Theta]\!]_\rho(w')$ by monotonicity. By assumption, we have, for all $v \in Val$, $[\![\Gamma, x : \mathsf{Val} \vdash \{P\}\ e\ \{Q\}]\!]_{\rho[x \mapsto v]}(w')$. Consequently,

$$\begin{aligned} m &\in [\![\Gamma \vdash \lambda r.\, \forall x : \mathsf{Val}.\, \mathsf{spec}(\{P\}\ rx\ \{Q\}) : \mathsf{Val} \to \mathsf{Spec}]\!]_\rho(w')(f) \\ &= \mathsf{wp}([\![\Gamma \vdash \lambda r.\, \forall x : \mathsf{Val}.\, \mathsf{spec}(\{P\}\ rx\ \{Q\}) : \mathsf{Val} \to \mathsf{Spec}]\!]_\rho)(f)(w') \end{aligned}$$

and so $\Gamma \mid \Theta \models \{\mathsf{emp}\}\ \lambda x.\, e\ \{r.\, \forall x : \mathsf{Val}.\, \mathsf{spec}(\{P\}\ rx\ \{Q\})\}$ as required.

Lemma 12 (SpecOut rule). *If*

$$\Gamma \mid \Theta, S \models \{P\}\ e\ \{R\}$$

then

$$\Gamma \mid \Theta \models \{P * \mathsf{spec}(S)\}\ e\ \{R\}$$

Proof. Let $\rho \in [\![\Gamma]\!]$ and $w \in \mathsf{W}$ be such that $[\![\Gamma \vdash \Theta]\!]_\rho(w)$. Let $m \in [\![\Gamma \vdash P * \mathsf{spec}(S)]\!]_\rho(w)$. Then $[\![\Gamma \vdash S]\!]_\rho(w)$ and $m \in [\![\Gamma \vdash P]\!]_\rho(w)$. Hence $[\![\Gamma \vdash \Theta, S]\!]_\rho(w)$, and thus by assumption

$$[\![\Gamma \vdash P : \mathsf{Prop}]\!]_\rho(w) \subseteq \mathsf{wp}([\![\Gamma \vdash Q : \mathsf{Val} \to \mathsf{Prop}]\!]_\rho)(\rho(e))(w)$$

It follows that $s \in \mathsf{wp}([\![\Gamma \vdash Q : \mathsf{Val} \to \mathsf{Prop}]\!]_\rho)(\rho(e))(w)$, as required.

Lemma 13 (Alloc rule).

$$\Gamma, v : \mathsf{Val} \mid \Theta \models \{\mathsf{emp}\}\ \mathsf{ref}\, v\ \{r.\, r \mapsto v\}$$

Proof. Let $\rho \in [\![\Gamma, v : \mathsf{Val}]\!]$ and $w \in \mathsf{W}$ be such that $[\![\Gamma, v : \mathsf{Val} \vdash \Theta]\!]_\rho(w)$. Let $m \in \mathsf{M}$. Suppose that $t, (h, \gamma) \vDash_w m \bullet r$ and $\mathsf{ref}\, \rho(v), (h, \gamma) \xrightarrow{a} l, (h', \gamma')$. Then it must be that $a = \epsilon$, $l \in \mathrm{Loc} \backslash \mathsf{dom}(h)$, $h' = h[l \mapsto \rho(v)]$ and $\gamma' = \gamma$. Consequently, $t, (h', \gamma') \vDash_w (m \bullet [l \mapsto \rho(v)]) \bullet r$. Now

$$\begin{aligned} m \bullet [x \mapsto \rho(v)] &\in [\![\Gamma, v : \mathsf{Val} \vdash \lambda r.\, r \mapsto v : \mathsf{Val} \to \mathsf{Prop}]\!]_\rho(l)(w) \\ &= \mathsf{wp}([\![\Gamma, v : \mathsf{Val} \vdash \lambda r.\, r \mapsto v : \mathsf{Val} \to \mathsf{Prop}]\!]_\rho)(l)(w) \end{aligned}$$

as required.

Lemma 14 (Read rule).

$$\Gamma, l, v : \mathsf{Val} \mid \Theta \models \{l \mapsto v\}\ !l\ \{r.\, l \mapsto v * r = v\}$$

Proof. Let $\rho \in [\![\Gamma, l, v : \mathsf{Val}]\!]$ and $w \in \mathsf{W}$ be such that $[\![\Gamma, l, v : \mathsf{Val} \vdash \Theta]\!]_\rho(w)$. Let $m \in [\![l \mapsto v]\!]_\rho(w)$. Suppose that $t, (h, \gamma) \vDash_w m \bullet r$ and $!\rho(l), (h, \gamma) \xrightarrow{a} e, (h', \gamma')$. (Note that since $h(\rho(l))) = (m \bullet r)(\rho(l)) = \rho(v)$, we do not have $!\rho(l), (h, \gamma) \not\rightarrow$.) It must be that $a = \epsilon$, $e = \rho(v)$, $h' = h$ and $\gamma' = \gamma$. Now

$$\begin{aligned} &m \in [\![\Gamma, l, v : \mathsf{Val} \vdash \lambda r.\, l \mapsto v * r = v : \mathsf{Val} \to \mathsf{Prop}]\!]_\rho(\rho(v))(w) \\ &= \mathsf{wp}([\![\Gamma, l, v : \mathsf{Val} \vdash \lambda r.\, l \mapsto v * r = v : \mathsf{Val} \to \mathsf{Prop}]\!]_\rho)(\rho(v))(w) \end{aligned}$$

as required.

Lemma 15 (Write rule).

$$\Gamma, l, v : \mathsf{Val} \mid \Theta \models \{l \mapsto \_\}\ l := v\ \{r.\, l \mapsto v * r = ()\}$$

Proof. Let $\rho \in [\![\Gamma, l, v : \mathsf{Val}]\!]$ and $w \in \mathsf{W}$ be such that $[\![\Gamma, l, v : \mathsf{Val} \vdash \Theta]\!]_\rho(w)$. Let $m \in [\![l \mapsto \_]\!]_\rho(w)$. Suppose that $t, (h, \gamma) \vDash_w m \bullet r$ and $(\rho(l) := \rho(v)), (h, \gamma) \to e, (h', \gamma')$. (Note that since $h(\rho(l)) = m(\rho(l)) = v_0$ for some $v_0 \in Val$, we do not have $(\rho(l) := \rho(v)), (h, \gamma) \not\rightarrow$.) It must be that $a = \epsilon$, $e = ()$, $h' = h[\rho(l) \mapsto \rho(v)]$ and $w' = w$. Let $m' = (h', \pi_2(m)) = (\pi_1(m)[\rho(l) \mapsto \rho(v)], \pi_2(m))$, so that $t, (h', \gamma) \vDash_w m'$ Now

$$\begin{aligned} &m' \in [\![\Gamma, l, v : \mathsf{Val} \vdash \lambda r.\, l \mapsto v * r = () : \mathsf{Val} \to \mathsf{Prop}]\!]_\rho(())(w) \\ &= \mathsf{wp}([\![\Gamma, l, v : \mathsf{Val} \vdash \lambda r.\, l \mapsto v * r = () : \mathsf{Val} \to \mathsf{Prop}]\!]_\rho)(())(w) \end{aligned}$$

as required.

Lemma 16 (Emit rule). *Let* $\Gamma' = \Gamma, t : \mathsf{seq\ Val}, v : \mathsf{Val}, I : \mathsf{seq\ Val} \to \mathsf{Bool}$. *If*

$$\Gamma' \mid \Theta \models t \cdot v \in I$$

then

$$\Gamma' \mid \Theta \models \{\mathsf{trace}(t) * \mathsf{inv}(I)\}\ \mathsf{emit}\ v\ \{r.\, \mathsf{trace}(t \cdot v) * r = ()\}$$

Proof. Let $\rho \in [\![\Gamma']\!]$ and $w \in \mathsf{W}$ be such that $[\![\Gamma' \vdash \Theta]\!]_\rho(w)$. Let $m \in [\![\mathsf{trace}(t) * \mathsf{inv}(I)]\!]_\rho(w)$. Suppose that $t_0, s \vDash_w m \bullet r$. It follows that $t_0 = \rho(t)$ and, by the assumption, $t_0 \cdot \rho(v) \in \pi_2(w)$. Suppose also that $\mathsf{emit}\ \rho(v), s \xrightarrow{a} e, s'$. (Note that we do not have $\mathsf{emit}\ \rho(v), (h, \gamma) \not\rightarrow$.) It must be that $a = \rho(v)$, $e = ()$ and $s' = s$. Let $m' = (\pi_1(m), \pi_2(m) \cdot \rho(v)) = (\pi_1(m), t_0 \cdot a)$. It must also be that $t_0 \cdot a \vDash_w m' \bullet r$. Now

$$\begin{aligned} &m' \in [\![\Gamma' \vdash \lambda r.\, \mathsf{trace}(t \cdot v) * r = () : \mathsf{Val} \to \mathsf{Prop}]\!]_\rho(())(w) \\ &= \mathsf{wp}([\![\Gamma' \vdash \lambda r.\, \mathsf{trace}(t \cdot v) * r = () : \mathsf{Val} \to \mathsf{Prop}]\!]_\rho)(())(w) \end{aligned}$$

as required.

Lemma 17. *Suppose that* $\Gamma \mid - \vdash \{P\}\ e\ \{Q\}$ *and let* $w \in \mathsf{W}, \rho \in [\![\Gamma]\!]$ *and* $m \in [\![\Gamma \vdash P]\!]_\rho(w)$. *Taking* r, t, s *such that* $t, s \vDash_w m$, *suppose that there exists* s', t' *with* $\rho(e), s \xrightarrow{t'}{}^{*} e', s' \not\rightarrow$. *Then* $e' \in Val$ *and there exists* $w' \geq w$ *and* $m' \in [\![\Gamma \vdash Q]\!]_\rho(e')(w')$ *such that* $(t \cdot t'), s' \vDash_{w'} m'$.

Proof. From Theorem 3, we get that $\Gamma \mid - \vDash \{P\}\ e\ \{Q\}$, i.e. $\forall w \in \mathsf{W}.\forall \rho \in [\![\Gamma]\!].[\![\Gamma \vdash \{P\}\ e\ \{Q\} : \mathsf{Spec}]\!]_\rho(w) = \top$. Thus, we get that $[\![\Gamma \vdash P : \mathsf{Prop}]\!]_\rho(w) \subseteq \mathsf{wp}([\![\Gamma \vdash Q : \mathsf{Val} \rightarrow \mathsf{Prop}]\!]_\rho)(\rho(e))(w)$. So taking m, r, t, s such that $m \in [\![\Gamma \vdash P]\!]_\rho(w)$ and $t, s \vDash_w m$, we know that $m \in \mathsf{wp}([\![\Gamma \vdash Q : \mathsf{Val} \rightarrow \mathsf{Prop}]\!]_\rho)(\rho(e))(w)$. Writing q for $[\![\Gamma \vdash Q : \mathsf{Val} \rightarrow \mathsf{Prop}]\!]_\rho$, from

- $m \in \mathsf{wp}(q)(\rho(e))(w)$
- $t, s \vDash_w m$
- $s, e \xrightarrow{t'}{}^{*} s', e'$

we prove by induction on the length of this reduction above that $e' \in Val$ and there exists $w' \geq w$ and $m' \in q(e')(w')$ such that $(t \cdot t'), s' \vDash_{w'} m'$.

For the base case, $s' = s$, $e' = \rho(e)$ and $t' = \varepsilon$. Unfolding the definition of wp, since $\rho(e), s \not\rightarrow$ it follows that $\rho(e) \in Val$ and that $m \in q(\rho(e))(w)$, so that we take $w' = w$ and $m' = m$.

For the inductive case, assume $\rho(e), s \xrightarrow{a} e'', s'' \xrightarrow{t'}{}^{n} e', s' \not\rightarrow$. Unfolding the definition of wp, it follows that there exists $w' \geq w$ and m' such that

$$t \cdot a, s'' \vDash_{w'} m' \qquad\qquad m' \in \mathsf{wp}(q)(e'')(w')$$

By the induction hypothesis, it follows that there exists $w'' \geq w'$ and $m'' \in q(e')(w'')$ such that $t \cdot a \cdot t', s' \vDash_{w''} m''$.

C Erasure of instrumentation

Lemma 18. *For all* $e, e', h, h', \gamma, \gamma'$, *if* $e, (h, \gamma) \rightarrow e', (h', \gamma')$ *(without emitting any events) then* $\widehat{e}, (h, \widehat{\gamma}) \rightarrow \widehat{e'}, (h', \widehat{\gamma'})$.

Proof. By case analysis on the reduction step, using the fact that $\widehat{K[e]} = \widehat{K}[\widehat{e}]$.

Theorem 1. *For all* $e, e', h, h', \gamma, \gamma'$, *if* $\widehat{e}, (h, \widehat{\gamma}) \rightarrow^* e', (h', \gamma')$ *then there exists a trace* t, *an expression* e'' *and an environment* γ'' *such that* $e' = \widehat{e''}, \gamma' = \widehat{\gamma''}$ *and* $e, (h, \gamma) \xrightarrow{t}{}^* e'', (h', \gamma'')$.

Proof. By induction on the (lexicographic order of the) length of the reduction of $\widehat{e}, (h, \widehat{\gamma}) \rightarrow^* e', (h', \gamma')$ and the number of emits in e. If $\widehat{e}, (h, \widehat{\gamma})$ is irreducible, then either:

- $e, (h, \gamma)$ is irreducible, so that we can define t as the empty trace, e'' as e and γ'' as γ.
- or $e = K[\mathsf{emit}\, u]$ so that $e, (h, \gamma) \xrightarrow{u} K[()], (h, \gamma)$. Then we can apply the induction hypothesis since $K[()]$ has strictly fewer emits than e, and $\widehat{e} = \widehat{K[()]}$.

Otherwise, there exist e_1, γ_1, h_1 such that $\widehat{e}, (\widehat{\gamma}, h) \to e_1, (\gamma_1, h_1) \to^* e', (\gamma', h')$. There are two possibilities:

- Either there exist $e_1', (\gamma_1', h_1')$, such that $e, (h, \gamma) \to e_1', (\gamma_1', h_1')$ without emitting any events, in which case, from the previous lemma and the determinism of the reduction, we get that $\widehat{e_1'} = e_1, \widehat{\gamma_1'} = \gamma_1$ and $h_1' = h_1$. We now use the IH to prove the claim.
- Or $e = K[\mathsf{emit}\ u]$ so that $e, (h, \gamma) \xrightarrow{u} K[()], (h, \gamma)$. Then we can apply the IH since $K[()]$ has strictly fewer emits than e, and $\widehat{e} = \widehat{K[()]}$.