A Concurrent Program Logic with a Future and History

ANONYMOUS AUTHOR(S)

Verifying fine-grained optimistic concurrent programs remains an open problem. Modern program logics provide abstraction mechanisms and compositional reasoning principles to deal with the inherent complexity. However, their use is mostly confined to pencil-and-paper or mechanized proofs. We devise a new separation logic geared towards the lacking automation. While local reasoning is known to be crucial for automation, we are the first to show how to retain this locality for (i) reasoning about inductive properties without the need for ghost code, and (ii) reasoning about computation histories in hindsight. We implemented our new logic in a tool and used it to automatically verify challenging concurrent search structures that require inductive properties and hindsight reasoning, such as the Harris set.

1 INTRODUCTION

Concurrency comes at a cost, at least, in terms of increased effort when verifying program correctness. There has been a proliferation of concurrent program logics that provide an arsenal of reasoning techniques to address this challenge [Bell et al. 2010; Delbianco et al. 2017; Elmas et al. 2010; Fu et al. 2010; Gotsman et al. 2013; Gu et al. 2018; Hemed et al. 2015; Jung et al. 2018; Liang and Feng 2013; Manna and Pnueli 1995; Parkinson et al. 2007; Sergey et al. 2015; Vafeiadis and Parkinson 2007]. In addition, a number of general approaches have been developed to help structure the high-level proof argument [Feldman et al. 2018, 2020; Kragl et al. 2020; O'Hearn et al. 2010; Shasha and Goodman 1988]. However, the use of these techniques has been mostly confined to manual proofs done on paper, or mechanized proofs constructed in interactive proof assistants. We distill from these works a concurrent separation logic suitable for automating the construction of local correctness proofs for highly concurrent data structures. We focus on concurrent search structures (sets and maps indexed by keys), but the developed techniques apply more broadly. Our guiding principle is to perform all inductive reasoning, both in time and space, in lock-step with the program execution. The reasoning about inductive properties of graph structures and computation histories is relegated to the meta-theory of the logic by choosing appropriate semantic models.

Running Example. We motivate our work using Harris' non-blocking set data structure [Harris 2001], which we will also use as a running example throughout the paper.

We assume a garbage-collected programming language, supporting (first-order) recursive functions, product types, and mutable heap-allocated structs. The language further provides a *compare-and-set* operation, `CAS(`x`.f,`o`,`n`)`, that atomically sets field `f` of x to n and returns `true` if `f`'s current value is o, or otherwise returns `false` leaving x`.f` unchanged.

Harris' algorithm implements a set data structure that takes elements from a totally ordered type `K` of keys and provides operations for concurrently finding, inserting, and deleting a given *operation key* `k`. We focus on the `find` operation shown in Figure 1. The data structure is represented as a linked list consisting of nodes implemented by the struct type `N`. Each node stores a `key` and a `next` pointer to the successor node in the list. A potential state of the data structure is illustrated in Figure 1. The algorithm maintains several important invariants. First, the list is strictly sorted by the keys in increasing order and has a sentinel head node, pointed to by the immutable shared pointer `head`. The key of the head node is $-\infty$. Likewise, there is a sentinel tail node with key ∞. We assume $-\infty < k < \infty$ for all operation keys k. To allow concurrent insertions and deletions without lock-based synchronization, a node that is to be removed from the list is first *marked* to indicate that it has been logically deleted before it is physically unlinked. Node marking is implemented by *bit-stealing* on the `next` pointers. We abstract from the involved low-level bit-masking using

```
1  struct N = { val key: K; var next: N }
2
3  val tail = new N { key = ∞; next = tail }
4  val head = new N { key = −∞; next = tail }
5
6  procedure traverse(k: K, l: N, ln: N, t: N): (N∗N∗N) {
7    val tn = t.next
8    val tmark = is_marked(tn)
9    if (tmark) return traverse(k, l, ln, tn)
10   else if (t.key < k) return traverse(k, t, tn, tn)
11   else return (l, ln, t)
12 }
```

```
13 procedure find(k: K): N ∗ N {
14   val hn = head.next
15   val l, ln, r := traverse(k, head, hn, hn)
16   if ((ln == r || CAS(l.next, ln, r))
17       && !is_marked(r.next)) return (l, r)
18   else return find(k)
19 }
20
21 procedure search(k: K) : Bool {
22   val _, r = find(k)
23   return r.key == k
24 }
```

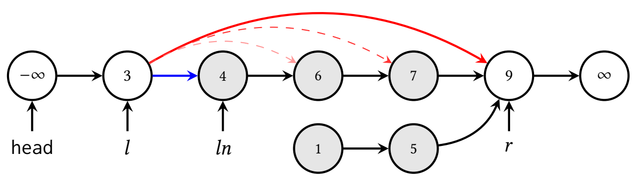

Fig. 1. The Harris set algorithm [Harris 2001]. The lower half shows a state of the Harris set containing keys $\{3, 9\}$. Nodes are labeled with the value of their key field. Edges indicate next pointers. Marked nodes are shaded gray. The blue edge between the nodes l and ln represents the state of l before the CAS and the red edge between l and r the state after the CAS from Line 16. Dashed edges represent the hypothetical updates that inductively capture the effect of the CAS.

the function `is_marked`. A call `is_marked(p)` returns `true` iff the mark bit of pointer p is set. We say that a node x is marked if `is_marked(x.next)` returns `true`. The sentinel nodes are never marked.

The workhorse of the algorithm is the function `find`. It takes an operation key k and returns a pair of nodes (l, r) such that l.key $< k \leq r$.key, and the following held true at a single point in time during `find`'s execution: r was unmarked and l's direct successor, and l was reachable from head. All client-facing functions such as `search` then use `find`.

Contributions. Recall that in order to prove linearizability of a concurrent data structure, one has to show that each of the data structure's operations takes effect instantaneously at some time point between its invocation and return, the *linearization point*, and behaves according to its sequential specification [Herlihy and Tygar 1987]. The Harris set exhibits two key challenges in automating linearizability proofs of non-blocking data structures that we aim to address in our work.

The first challenge is that linearization points may not be statically fixed, but instead depend on the interference of concurrent operations performed by other threads. We discuss this issue using `search`, whose sequential specification says that the return value is `true` iff k is present in the data structure. Consider a thread T that executes a call `search(7)`. Figure 1 shows a possible intermediate state of the list observed right after the successful execution of the CAS on Line 16 during T's execution of `search`. This is also the linearization point of `search` for T: the key 7 is currently not present in the data structure and `search` will return `false` because the condition r`.key == 7` on Line 23 will evaluate to `false` regardless of what other threads do, such as inserting 7 before `search(7)` returns. However, in an alternative execution, a concurrently executing `find` may precede T in executing the CAS that sets l`.next` to r so that by the time T executes Line 16, the condition $ln == r$ is true. In this situation, the operation linearized when the pointer ln was read (i.e., on Line 7 or 14). In general, the correct linearization point may only be known in hindsight [O'Hearn et al. 2010] at a later point in the execution. As a consequence, the proof must track information about earlier states in the execution history to enable reasoning about linearization points that already happened in the past.

Our first contribution is a lightweight embedding of computation histories into separation logic [O'Hearn et al. 2001] that supports local proofs using hindsight arguments, but without having to perform explicit induction over computation histories.

The second challenge is that the proof needs to reason about maintenance operations that are local but affect an unbounded heap region. The procedure `traverse` guarantees that all nodes between l and r are marked. The `CAS` on Line 16 then unlinks the segment of marked nodes between l and r from the structure making r the direct successor of l. This step is depicted in Figure 1. The blue edge between l and ln refers to the pre state of the `CAS` and the red edge between l and r to the post state. A traditional automated analysis needs to infer the precise inductive shape invariant about the traversed region (e.g. a recursive predicate stating that it is a list segment of marked nodes). Then, at the point where the segment is unlinked, it has to infer that the global data structure invariant is maintained. This involves an inductive proof argument, and the analysis needs to rediscover how this induction relates to the invariant of the traversal.

Our second contribution is a mechanism for reasoning about such updates with non-local effects. The idea is to compose these updates out of *ghost update chunks*. This is illustrated in Figure 1, where the effect of the `CAS` is composed out of simpler updates that move the edge from l towards r one node at a time (indicated by the dashed red edges). One only needs to reason about four nodes to prove that the edge can be moved forward by one node. We refer to a correctness statement of such a ghost update chunk as a *future*. The crux is to construct these futures during the traversal of the marked segment, i.e., in lock-step with the program execution. This avoids the need for explicit inductive reasoning at the point where the `CAS` takes effect. When proving the future for unlinking a single traversed node t, we directly apply interference-free facts learned during the traversal, e.g., that t must be marked and can therefore be unlinked. We call this mechanism accounting. The final future can then be invoked on Line 16 to prove the correctness of the `CAS`.

The focus of the paper is on the development of the new program logic rather than algorithmic details on efficient automatic proof search. However, we have implemented a prototype tool called `plankton` that uses the logic to automatically verify linearizability of non-blocking concurrent search structures. The tool's implementation follows a standard abstract interpretation approach [Cousot and Cousot 1977]. We have successfully applied the tool to verify several fine-grained non-blocking and lock-based concurrent set implementations, including: Harris set [Harris 2001], Michael set [Michael 2002], Vechev and Yahav CAS sets [Vechev and Yahav 2008, Figure 2], ORVYY set [O'Hearn et al. 2010], and the Lazy set [Heller et al. 2005]. All these benchmarks require hindsight reasoning. To our knowledge, `plankton` is the first tool that automates hindsight reasoning for such a variety of benchmarks. The Harris set additionally requires futures that are also automated in plankton. With this, `plankton` is the first tool that can automatically verify the Harris set algorithm.

2 OVERVIEW

We aim for a proof strategy that is compatible with local reasoning principles and agnostic to the detailed invariants of the specific data structure under consideration. In particular, we want to avoid proof arguments that devolve into explicit reasoning about heap reachability or other inductive heap properties, which tend to be difficult to automate.

Our strategy builds on the *keyset framework* [Krishna et al. 2020a; Shasha and Goodman 1988] for designing and verifying concurrent search structures. In this framework, the data structure's heap graph is abstracted by a mathematical graph (N, E) where each node $x \in N$ is labeled by its local contents, a set of keys $\mathsf{C}(x)$. The abstract state of the data structure $\mathsf{C}(N)$ is then given by the union of all node-local contents. Moreover, each node x has an associated set of keys $\mathsf{KS}(x)$ called its *keyset*. The keysets are defined inductively over the graph such that the following *keyset invariants* are maintained: (1) the keysets of all nodes partition the set of all keys, and (2) for all nodes x, $\mathsf{C}(x) \subseteq \mathsf{KS}(x)$. For the Harris set, we define $\mathsf{C}(x) \triangleq \mathit{mark}(x)\ ?\ \emptyset : \{\, \mathit{key}(x) \,\}$ and let $\mathsf{KS}(x)$ be the empty set if x is not reachable from `head` and otherwise the interval $(\mathit{key}(y), \mathit{key}(x)]$ where y is the predecessor of x in the list (cf. Figure 2). Here, we denote by $\mathit{key}(x)$ the value of field `key` in a given state, and similarly for $\mathit{mark}(x)$. Throughout the rest of the paper, we will follow this convention of naming variables in a way that reflects the referencing mechanism.

The keyset invariants ensure that for any node $x \in N$ and key k

$$k \in \mathsf{KS}(x) \Rightarrow (k \in \mathsf{C}(N) \Leftrightarrow k \in \mathsf{C}(x)) \enspace .$$

This property allows us to reduce the correctness of an insertion, deletion, and search for k on the global abstract state $\mathsf{C}(N)$ to the correctness of the same operation on x's local contents $\mathsf{C}(x)$, provided we can show $k \in \mathsf{KS}(x)$. For example, suppose that a concurrent invocation of `search`(k) returns `false`. To prove that this invocation is linearizable, it suffices to show that there exists a node x such that both $k \in \mathsf{KS}(x)$ and $k \notin \mathsf{C}(x)$ were true at the same point in time during `search`(k)'s execution. We refer to x as the decisive node of the operation. The point in time where the two facts about x hold is the linearization point.

The ingredients for the linearizability proofs are thus (i) defining the keysets for the data structure at hand, (ii) proving that the keyset invariants are maintained by the data structure's operations, and (iii) identifying the linearization point and decisive node for an operation on key k by establishing the relevant facts about k's membership in the keyset and contents of the decisive node.

Our contributions focus on the automation of (ii) and (iii). While we do not automate (i), the definition of the keyset follows general principles and can be reused across many data structures [Shasha and Goodman 1988] (e.g., we use the same definition for all the list-based set implementations considered in our evaluation, cf. §8). In the remainder of this section, we provide a high-level overview of the reasoning principles that underlie our new program logic and enable proof automation.

Automating History Reasoning. We start with the linearizability argument. Consider a concurrent execution of `search`(k) that returns value b. The decisive node of `search` is always the node r returned by the call to `find`. The proof thus needs to establish that at some point during the execution, $k \in \mathsf{KS}(r)$ and $b \Leftrightarrow (k \in \mathsf{C}(r))$ were true. The issue is that when we reach the corresponding point in the execution during proof construction, we may not be able to linearize the operation right away because the decisive node and linearization point depend on the interferences by other threads that may still occur in the remainder of the execution. We can thus only linearize the execution in hindsight, once all the relevant interferences have been observed.

To illustrate this point, consider the scenario depicted in Figure 2. It shows intermediate states of two executions of `search(5)` on a Harris set that initially contains the keys $\{2, 5\}$. The two executions agree up to the point when state s_1 is reached at the beginning of Line 7 in the first call to `traverse` (i.e., after n_2's `next` pointer and mark bit have been read). The execution depicted on the bottom proceeds without interference to state s_2 at the beginning of Line 17 and will return $b = \mathit{true}$. Here, the decisive node is n_3 and the linearization point is s_1. Note that both $5 \in \mathsf{KS}(n_3)$ and $5 \in \mathsf{C}(n_3)$ hold in s_1. On the other hand, the execution depicted on the top of Fig. 2 is interleaved with a concurrently executing `delete`(5). The `delete` thread marks n_3 before the `search` thread reaches the beginning of Line 17, yielding state s'_2 at this point. The test whether r is marked will now fail, causing the `search` thread to restart. After traversing the list again, the `search` thread will unlike n_3 from the list with the `CAS` on Line 16. This yields state s'_3 when the `search` reaches Line 17 again. The `search` thread will then proceed to compute the return value $b = \mathit{false}$. For this execution, the decisive node is n_4 and the linearization point is s'_3.

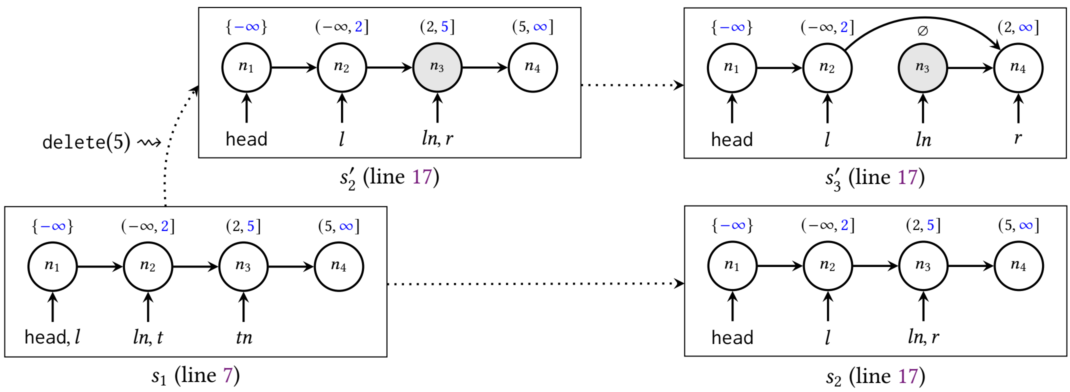

Fig. 2. Some states observed during two executions of `search(5)` on a Harris set that initially contains the keys 2 and 5 and no marked nodes. Each node in a state is labeled above by its keyset. If the node is reachable from `head`, the right bound of the keyset interval indicates the key stored in the node (highlighted in blue).

Our program logic provides two ingredients for dealing with the resulting complexity in the linearizability proof. We discuss these formally in §7. The first ingredient is the *past predicate* $\diamondminus p$, which asserts that the current thread owned the resource p at some point in the past. The second ingredient is a set of proof rules for introducing and manipulating past predicates. In particular, we will use the following three rules in our proof:

$$\textsc{h-intro}\quad \vdash \{\, p \,\}\ \mathtt{skip}\ \{\, p * \diamondminus p \,\} \qquad \textsc{h-hindsight}\ \frac{p \text{ pure}}{p * \diamondminus q \vdash \diamondminus(p * q)} \qquad \textsc{h-infer}\ \frac{p \vdash q}{\diamondminus p \vdash \diamondminus q}$$

The rule h-intro states the validity of the Hoare triple $\{\, p \,\}$ `skip` $\{\, p * \diamondminus p \,\}$ which introduces a past predicate $\diamondminus p$ using a *stuttering step*. Here, $*$ is separating conjunction. The rule expresses that if the thread owns p now, it trivially owned p at some past point up until now. The rule h-hindsight captures the essence of hindsight reasoning: ownership of p can be transferred from the now into the past, if p is a purely logical fact that is independent of the program state. The rule h-infer states the monotonicity of the past operator with respect to logical weakening.

We demonstrate the use of past predicates and their associated rules by sketching the linearizability proof for executions of `search`(k) that follow the same code path as the one in the bottom half of Figure 2. The proof relies on a predicate $\mathsf{Node}(x)$. This predicate expresses ownership of the physical representation of x in the heap and binds the *logical variables* $key(x)$ and $mark(x)$ to the values stored in the relevant fields of x. The predicate also expresses important properties needed for maintaining the keyset invariants. §5 discusses the definition of the predicate $\mathsf{Node}(x)$ in detail.

The proof proceeds by symbolic execution of the considered path. The goal is to infer

$$\diamondminus(`\mathsf{Node}(r) * k \in `\mathsf{KS}(r) \wedge (b \Leftrightarrow k \in `\mathsf{C}(r))$$

as post condition where b is the return value of `search`(k). This implies the existence of a linearization point as discussed earlier. Here, we write $`e$ for the expression obtained from expression $e$ by replacing all logical variables like $mark(r)$ by fresh variables $`mark(r)$. That is, $`e$ can be thought of as the expression $e$ evaluated with respect to the old state of $r$ captured by $`\mathsf{Node}(r)$ inside the past predicate, rather than the current state.

The symbolic execution starts from a global shared invariant that maintains $\mathsf{Node}(x)$ for all nodes $x \in N$. When symbolic execution reaches Line 7 in the proof, we can establish shared ownership of $\mathsf{Node}(t)$ and $\mathsf{Node}(tn)$ using the derived invariant of the traversal. The two predicates imply that if

t is unmarked, then its keyset must be non-empty. In turn, this implies $\mathsf{KS}(tn) = (key(t), key(tn)]$, because tn is the successor of t. Together, we deduce (a) $\neg mark(t) \wedge key(t) < k \wedge k \leq key(tn) \Rightarrow k \in \mathsf{KS}(tn)$. Moreover, the definition of $\mathsf{C}(tn)$ gives us (b) $k \in \mathsf{C}(tn) \Leftrightarrow \neg mark(tn) \wedge key(tn) = k$. We let $H(tn, t)$ be the conjunction of (a), (b), and $\mathsf{Node}(tn)$.

Next, we use rule H-INTRO to transfer $H(tn, t)$ into a past predicate, yielding $H(tn, t) * \diamondminus(H(tn, t))$. For our proof to be valid, we need to make sure that all intermediate assertions are stable under interferences by other threads. Unfortunately, this is not the case for the assertion $H(tn, t) * \diamondminus(H(tn, t))$. Notably, this assertion implies that the current value of tn's mark bit is the same as the value of its mark bit in the past state referred to by the past predicate.

To make the assertion stable under interference, we first introduce fresh logical variables $\backprime key(tn)$ and $\backprime mark(t)$ which we substitute for $key(tn)$ and $mark(tn)$ under the past operator. This yields the equivalent intermediate assertion:

$$H(tn, t) * \diamondminus(\backprime H(tn, t)) \; * \; (\backprime mark(tn) = mark(tn) \wedge \backprime key(tn) = key(tn)) \;\; .$$

Next, we observe that other threads executing `search`, `insert`, and `delete` operations can only interfere by marking node tn in case it is not yet marked. Such interference invalidates the equality $mark(tn) = \backprime mark(tn)$. So we weaken it to $\backprime mark(tn) \Rightarrow mark(tn)$. In §4 we introduce a general Owicki-Gries-style separation logic framework that formalizes this form of interference reasoning. In addition, we keep only $\mathsf{Node}(tn)$ from $H(tn, t)$, leaving us with the interference-free assertion

$$P(tn, t) \triangleq \mathsf{Node}(tn) * \diamondminus(\backprime H(tn, t)) \; * \; (\backprime mark(tn), t \Rightarrow mark(tn), t) \wedge \backprime key(tn) = key(tn) \;\; .$$

We then propagate this assertion forward along the considered execution path of `search`(k), obtaining $P(ln, t)$ when Line 17 is reached in the proof. During the propagation, we accumulate the facts $\neg mark(t)$, $key(t) < k$, $k \leq key(r)$, and $ln = r$ according to the branches of the conditional expressions taken along the path. As these facts are all pure, we use rule H-HINDSIGHT to transfer them, together with the equality $\backprime key(ln) = key(ln)$, inside the past predicate $\diamondminus(\backprime H(ln, t))$.

Using the rule H-INFER we can then simplify the resulting past predicate as follows:

$$\diamondminus(\backprime\mathsf{Node}(r) \; * \; k \in \backprime\mathsf{KS}(r) \wedge (k \in \backprime\mathsf{C}(r) \Leftrightarrow \neg\backprime mark(r) \wedge \backprime key(r) = k))$$

As we propagate the overall assertion further to the return point of `search`(k), we accumulate the additional pure facts $\neg mark(r)$ and $b \Leftrightarrow key(r) = k$. We again invoke rule H-HINDSIGHT to transfer these into the past predicate, together with $\backprime mark(r) \Rightarrow mark(r)$ and $\backprime key(r) = key(r)$. The resulting past predicate can then be simplified with H-INFER to finally obtain the desired:

$$\diamondminus(\backprime\mathsf{Node}(r) * k \in \backprime\mathsf{KS}(r) \wedge (b \Leftrightarrow k \in \backprime\mathsf{C}(r)) \;\; .$$

While this proof is non-trivial, it is easy to automate. The analysis performs symbolic execution of the code. After each atomic step, it applies the rules H-INTRO and H-HINDSIGHT eagerly. The resulting past predicates are then simplified and weakened with respect to interferences by other threads. This way, the analysis maintains the strongest interference-free information about the history of the computation. These steps are then integrated into a classical fixpoint computation to infer loop invariants, applying standard widening techniques to enforce convergence [Cousot and Cousot 1979].

It is worth pointing out that the above reasoning could also be done with prophecies [Jung et al. 2020; Liang and Feng 2013]. However, prophecies are not amenable to automation in the same way as past predicates. The main reason for this is the hindsight rule: it works relative to facts that have already been discovered during symbolic execution. Prophecies would require to *guess* the same facts prior to being discovered. This guessing step is notoriously hard to automate [Bouajjani et al. 2017].

Automating Future Reasoning. Our second major contribution is the idea of futures and their governing reasoning principles. We motivate futures with the problem of automatically proving that the Harris set maintains the keyset invariants. As noted earlier, this is challenging because the

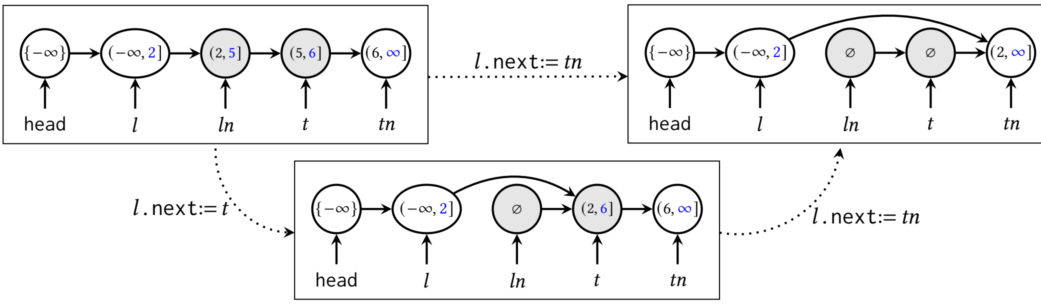

Fig. 3. A compound update (top) and its decomposition into update chunks (bottom). Compound updates may affect an unbounded number of nodes, e.g., changing their keysets. Update chunks allow to localize this effect of an update to a small, bounded number of nodes.

`CAS` performed by `find` may unlink unboundedly many marked nodes. Unlinking a node changes its keyset to the empty set. Hence, the `CAS` may affect an unbounded heap region. Showing that the `CAS` maintains the keyset invariants therefore inevitably involves an inductive proof argument.

Intuitively, we can reason about the effect of the `CAS` that unlinks the marked segment by composing it out of a sequence of *update chunks* that unlink the nodes in the segment one by one as indicated in Figure 1. This yields an inductive proof argument where we reason about simpler updates that only affect a bounded number of nodes at a time. The correctness of an update chunk `com` is represented by a future $\langle P \rangle\, \texttt{com}\, \langle Q \rangle$. A future can be thought of as a Hoare triple with precondition P and postcondition Q. However, futures inhabit the assertion level of the logic, rather than the meta level. The crux of our program logic is that it allows one to derive a future *on the side* in a subproof, while proving the correctness of the thread's traversal of the marked segment. The advantage of this approach is that it allows one to reuse the loop invariant for the proof of `traverse` towards proving the correctness of $\langle P \rangle\, \texttt{com}\, \langle Q \rangle$. This aids proof automation: the analysis no longer needs to synthesize the induction hypothesis out of thin air at the point where the `CAS` is executed. In a way, `traverse` moonlights as ghost code that aids the correctness proof of the `CAS`.

At the point where the `CAS` is executed, the proof then invokes the constructed future. A future can thus be thought of as a subproof that is saved up to be applied at some future point.

This idea is illustrated in Figure 3. The top left of the figure shows a state that is reached during the traversal of the marked segment from l to tn. The transition at the top depicts the effect of the update $l.\texttt{next} := tn$ applied to this state. The update unlinks the marked nodes between l and tn in one step. The correctness of this update is expressed by the future

$$F \triangleq \langle\, P * \mathit{next}(l) = \mathit{ln} * \mathit{mark}(\mathit{ln}) \,\rangle\ l.\texttt{next} := \mathit{tn}\ \langle\, P * \mathit{next}(l) = \mathit{tn} \,\rangle$$

where P is an appropriate invariant holding the relevant physical resources of the involved nodes. We derive this future by composing two futures for simpler update chunks as depicted at the bottom half of Figure 3. The left update chunk is described by the future

$$F_1 \triangleq \langle\, P * \mathit{next}(l) = \mathit{ln} * \mathit{mark}(\mathit{ln}) \,\rangle\ l.\texttt{next} := t\ \langle\, P * \mathit{next}(l) = t \,\rangle$$

and the right update chunk is described by

$$F_2 \triangleq \langle\, P * \mathit{next}(l) = t * \mathit{mark}(t) * \mathit{next}(t) = \mathit{tn} \,\rangle\ l.\texttt{next} := \mathit{tn}\ \langle\, P * \mathit{next}(l) = \mathit{tn} \,\rangle \quad .$$

F_1 is derived inductively during the traversal of the marked segment from l to t using the same process that we are about to describe for deriving F. The future F_2 can be easily proved in isolation. In particular the precondition $\mathit{mark}(t)$ implies $\mathsf{C}(t) = \emptyset$. Hence, the keyset invariant $\mathsf{C}(t) \subseteq \mathsf{KS}(t)$

is maintained. The condition $next(t) = tn$ guarantees that the update does not affect keysets of other nodes beyond t and tn.

We would now like to compose F_1 and F_2 using the standard sequential composition rule of Hoare logic to derive the future

$$F' \triangleq \langle\, P * next(l) = ln * mark(ln) \,\rangle \;\; l.\mathtt{next} := t; l.\mathtt{next} := tn \;\; \langle\, P * next(l) = tn \,\rangle \quad .$$

Once we have F', we get F by replacing the command $l.\mathtt{next} := t; l.\mathtt{next} := tn$ in F' with $l.\mathtt{next} := tn$ using a simple subsumption argument.

However, sequential composition requires that the postcondition of F_1 implies the precondition of F_2. Unfortunately, the precondition of F_2 makes the additional assumptions $mark(t)$ and $next(t) = tn$ that are not guaranteed by F_1. Now, observe that both of these facts are readily available in the outer proof context of the traversal: $next(t) = tn$ follows from Line 7 of `traverse` and $mark(t)$ is obtained from the condition on Line 9. We can transfer these facts from the outer proof context into F_2. This eliminates them from the precondition and enables the sequential composition to obtain F'. We refer to this transfer of facts as *accounting*. In a concurrent setting, accounting is sound provided the accounted facts are interference-free. This is the case here since the `next` fields of marked nodes are never changed and marked nodes are never unmarked. We explain these reasoning steps in more detail in §6.

The idea of futures applies more broadly beyond this specific example. They are useful whenever complex updates are prepared in advance by a traversal phase (as e.g. in Bw trees [Levandoski et al. 2013] and skip lists [Fraser 2004]).

3 PROGRAMMING MODEL

We study concurrency libraries where an unbounded number of threads executes the same program code. Our development is parametric in the set of states and set of commands, following the approach of abstract separation logic [Calcagno et al. 2007; Dinsdale-Young et al. 2013; Jung et al. 2018].

States. We assume that states form a *separation algebra*, a partial commutative monoid $(\Sigma, *, \mathsf{emp})$. For the set of units $\mathsf{emp} \subseteq \Sigma$ we require that (i) for all $\mathsf{s} \in \Sigma$, there exists some $1_\mathsf{s} \in \mathsf{emp}$ with $\mathsf{s} * 1_\mathsf{s} = \mathsf{s}$ and (ii) for all distinct $1, 1' \in \mathsf{emp}$, $1 * 1'$ is undefined. We use $\mathsf{s}_1 \# \mathsf{s}_2$ to indicate definedness.

Predicates are sets of states $p \in \mathbb{P}(\Sigma)$. In the appendix, we introduce an assertion language to denote predicates but to simplify the exposition we stay on the semantic level. Predicates form a Boolean algebra $(\mathbb{P}(\Sigma), \cup, \cap, \subseteq, \overline{}, \emptyset, \Sigma)$. *Separating conjunction* $p * q$ lifts the composition from states to predicates. This yields a commutative monoid with unit emp. *Separating implication* $p \mathrel{-\!\!*} q$ gives residuals:

$$p * q \;\triangleq\; \{\, \mathsf{s}_1 * \mathsf{s}_2 \;\mid\; \mathsf{s}_1 \in p \,\wedge\, \mathsf{s}_2 \in q \,\wedge\, \mathsf{s}_1 \# \mathsf{s}_2 \,\} \quad \text{and} \quad p \mathrel{-\!\!*} q \;\triangleq\; \{\, \mathsf{s} \;\mid\; \{\mathsf{s}\} * p \;\subseteq\; q \,\} \;.$$

In our development, $(\Sigma, *, \mathsf{emp})$ is a product of two separation algebras $(\Sigma_\mathsf{G}, *_\mathsf{G}, \mathsf{emp}_\mathsf{G})$ and $(\Sigma_\mathsf{L}, *_\mathsf{L}, \mathsf{emp}_\mathsf{L})$. We require $\mathsf{emp} \triangleq \mathsf{emp}_\mathsf{G} \times \mathsf{emp}_\mathsf{L} \subseteq \Sigma \subseteq \Sigma_\mathsf{G} \times \Sigma_\mathsf{L}$. In addition, Σ must be closed under decomposition: if $(\mathsf{g}_1 *_\mathsf{G} \mathsf{g}_2, \mathsf{l}_1 *_\mathsf{L} \mathsf{l}_2) \in \Sigma$ then $(\mathsf{g}_1, \mathsf{l}_1) \in \Sigma$, for all $\mathsf{g}_1, \mathsf{g}_2 \in \Sigma_\mathsf{G}$ and $\mathsf{l}_1, \mathsf{l}_2 \in \Sigma_\mathsf{L}$.

For $(\mathsf{g}, \mathsf{l}) \in \Sigma$ we call g the *global state* and think of it as the shared heap, although our development does not rely on this understanding. The component l is the *local state*, and we think of it as a thread's stack holding pointers through which to access the heap. States are composed component-wise, $(\mathsf{g}_1, \mathsf{l}_1) * (\mathsf{g}_2, \mathsf{l}_2) \triangleq (\mathsf{g}_1 *_\mathsf{G} \mathsf{g}_2, \mathsf{l}_1 *_\mathsf{L} \mathsf{l}_2)$, and this composition is defined only if the resulting product is again in Σ.

Lemma 1. *$(\Sigma, *, \mathsf{emp})$ is a separation algebra.*

Commands. The second parameter to our development is the set of commands $(\mathsf{COM}, [\![-]\!])$ that may be used to modify states. The set may be infinite, which allows us to treat atomic blocks

$$\mathtt{com} \xrightarrow{\mathtt{com}} \mathtt{skip} \qquad \mathtt{skip};\mathtt{st} \xrightarrow{\mathtt{skip}} \mathtt{st} \qquad \mathtt{st}^* \xrightarrow{\mathtt{skip}} \mathtt{skip} + \mathtt{st};\mathtt{st}^*$$

$$\frac{i \in \{1,2\}}{\mathtt{st}_1 + \mathtt{st}_2 \xrightarrow{\mathtt{skip}} \mathtt{st}_i} \qquad \frac{\mathtt{st}_1 \xrightarrow{\mathtt{com}} \mathtt{st}_1'}{\mathtt{st}_1;\mathtt{st}_2 \xrightarrow{\mathtt{com}} \mathtt{st}_1';\mathtt{st}_2} \qquad \frac{\mathtt{st}_1 \xrightarrow{\mathtt{com}} \mathtt{st}_2 \qquad (\mathsf{g}_2,\mathsf{l}_2) \in [\![\mathtt{com}]\!](\mathsf{g}_1,\mathsf{l}_1)}{(\mathsf{g}_1, \mathsf{pc}[i \mapsto (\mathsf{l}_1, \mathtt{st}_1)]) \to (\mathsf{g}_2, \mathsf{pc}[i \mapsto (\mathsf{l}_2, \mathtt{st}_2)])}$$

Fig. 4. Transition relation $\to\, \subseteq \mathsf{CF} \times \mathsf{CF}$ based on the control-flow relation $\to\, \subseteq \mathsf{ST} \times \mathsf{COM} \times \mathsf{ST}$.

as single commands. The effect of the commands on the states is defined by an interpretation. It assigns to each command a non-deterministic state transformer $[\![\mathtt{com}]\!]$ that takes a state and returns the set of possible successor states, $[\![\mathtt{com}]\!] : \Sigma \to \mathbb{P}(\Sigma)$. We lift the state transformer to predicates [Dijkstra 1976] in the expected way, $[\![\mathtt{com}]\!](p) \triangleq \bigcup_{s \in p} [\![\mathtt{com}]\!](s)$. We assume to have a command skip interpreted as the identity. For the frame rule to be sound, we expect the following monotonicity to hold [Calcagno et al. 2007; Dinsdale-Young et al. 2013], for all p, q, o:

$$[\![\mathtt{com}]\!](p) \subseteq q \quad \text{implies} \quad [\![\mathtt{com}]\!](p * o) \subseteq q * o\,. \tag{LocCom}$$

We handle commands that admit faults as in [Calcagno et al. 2007] by letting them return abort which is added as a new top element to the powerset lattice $\mathbb{P}(\Sigma)$, so that (LocCom) is trivially satisfied.

Concurrency libraries. Having fixed the set of states and the set of commands, a concurrency library is defined by a single program that is executed in every thread. The assumption of a single program can be made without loss of generality. The program code is drawn from the standard while-language ST defined by:

$$\mathtt{st} \ ::= \ \mathtt{com} \ \mid \ \mathtt{st} + \mathtt{st} \ \mid \ \mathtt{st};\mathtt{st} \ \mid \ \mathtt{st}^*\,.$$

The semantics of the library is defined in terms of unlabeled transitions among configurations. A *configuration* is a pair $\mathsf{cf} = (\mathsf{g}, \mathsf{pc})$ consisting of a global state $\mathsf{g} \in \Sigma_\mathsf{G}$ and a program counter $\mathsf{pc} : \mathbb{N} \to \Sigma_\mathsf{L} \times \mathsf{ST}$. The program counter assigns to every thread, modeled as a natural number, the current local state and the statement to be executed next. We use CF to denote the set of all configurations. A configuration $(\mathsf{g}, \mathsf{pc})$ is *initial* for predicate p and library code st, if the program counter of every thread yields a local state $(\mathsf{l}, \mathtt{st})$ where the code is the given one and the state satisfies $(\mathsf{g}, \mathsf{l}) \in p$. The configuration is *accepting* for predicate q, if every terminated thread $(\mathsf{l}, \mathtt{skip})$ satisfies the predicate, $(\mathsf{g}, \mathsf{l}) \in q$. We write these configuration predicates as the following sets

$$\mathsf{Init}_{p,\mathtt{st}} \triangleq \{(\mathsf{g},\mathsf{pc}) \mid \forall i, \mathsf{l}, \widehat{\mathtt{st}}.\, \mathsf{pc}(i) = (\mathsf{l}, \widehat{\mathtt{st}}) \Rightarrow (\mathsf{g},\mathsf{l}) \in p \wedge \widehat{\mathtt{st}} = \mathtt{st}\}$$
$$\mathsf{Acc}_q \triangleq \{(\mathsf{g},\mathsf{pc}) \mid \forall i, \mathsf{l}.\ \mathsf{pc}(i) = (\mathsf{l}, \mathtt{skip}) \Rightarrow (\mathsf{g},\mathsf{l}) \in q\}\,.$$

The unlabeled transition relation among configurations is defined in Figure 4. It relies on a labeled transition relation capturing the flow of control. A command may change the global state and the local state of the executing thread. It will not change the local state of other threads. A computation of the library is a finite sequence of consecutive transitions. A configuration is reachable if there is a computation that leads to it. We write $\mathsf{Reach}(\mathsf{cf})$ for the set of all configurations reachable from cf and lift the notation to sets where needed.

4 OWICKI-GRIES FOR CONCURRENCY LIBRARIES

We formulate the correctness of concurrency libraries as the validity of Hoare triples $\{\,p\,\}\ \mathtt{st}\ \{\,q\,\}$. A Hoare triple is valid if for every configuration cf that is initial wrt. p and st, every reachable configuration cf' is accepting wrt. q. Note that the definition does not refer to a single but to all threads executing the library code.

Definition 2. $\models \{\,p\,\}\ \mathtt{st}\ \{\,q\,\} \triangleq \mathsf{Reach}(\mathsf{Init}_{p,\mathtt{st}}) \subseteq \mathsf{Acc}_q$.

To establish this validity, we develop a thread-modular reasoning principle [Owicki and Gries 1976] that proceeds in two steps. First, we verify the library code as if it was run by an isolated thread using judgments that take the form $\mathbb{P}, \mathbb{I} \Vdash \{ p \}\ \mathtt{st}\ \{ q \}$.

The Hoare triple of interest is augmented by two pieces of information. The set $\mathbb{P}$ contains the intermediary predicates encountered during the proof of the isolated thread. The set $\mathbb{I}$ contains the interferences, the changes the isolated thread may perform on the shared state. The notion of interference will be made precise in a moment. Recording both sets during the proof is an idea we have taken from [Dinsdale-Young et al. 2013, Section 7.3].

The second phase of the thread-modular reasoning is to check that the local proof still holds in the presence of other threads. This is the famous interference-freedom check. It takes as input the computed sets $\mathbb{P}$ and $\mathbb{I}$ and verifies that no interference can invalidate a predicate, denoted by $\boxplus_{\mathbb{I}} \mathbb{P}$.

Interference. An *interference* is a pair (o, com) consisting of a predicate and a command. It represents the fact that from states in o environment threads may execute command com. A state (g, l) held by the isolated thread of interest will change under the interference to a state in

$$[\![(o, \mathsf{com})]\!](\mathsf{g}, \mathsf{l}) \;\triangleq\; \{ (\mathsf{g}', \mathsf{l}) \;\mid\; \exists \mathsf{l}_1, \mathsf{l}_2.\;\; (\mathsf{g}, \mathsf{l}_1) \in o \;\wedge\; (\mathsf{g}', \mathsf{l}_2) \in [\![\mathsf{com}]\!](\mathsf{g}, \mathsf{l}_1) \} \,.$$

We consider every state $(\mathsf{g}, \mathsf{l}_1) \in o$ that agrees with (g, l) on the global component, compute the post, and combine the resulting global component with the local component l. The agreement of different threads on the global state is precisely what is used in program logics like RGSep [Vafeiadis 2008; Vafeiadis and Parkinson 2007]. We lift $[\![(o, \mathsf{com})]\!]$ to predicates in the expected way.

We only record interferences that have an effect on the global state. An interference (o, com) is *effectful*, denoted by $\mathsf{eff}(o, \mathsf{com})$, if it changes the shared state of an element in o:

$$\mathsf{eff}(o, \mathsf{com}) \;\triangleq\; \exists (\mathsf{g}_1, \mathsf{l}_1) \in o. \exists (\mathsf{g}_2, \mathsf{l}_2) \in [\![\mathsf{com}]\!](\mathsf{g}_1, \mathsf{l}_1). \mathsf{g}_2 \neq \mathsf{g}_1 \,.$$

The thread-local proof computes a set of interferences. For a predicate o and a command com, the interference set is $\mathsf{inter}(o, \mathsf{com}) \triangleq \{(o, \mathsf{com})\}$ if $\mathsf{eff}(o, \mathsf{com})$ and $\mathsf{inter}(o, \mathsf{com}) \triangleq \varnothing$ otherwise. We consider interference sets up to the operation of joining predicates for the same command, $\{(p, \mathsf{com})\} \cup \{(q, \mathsf{com})\} = \{(p \cup q, \mathsf{com})\}$. Then $\mathsf{inter}(o, \mathsf{com}) \subseteq \mathbb{I}$ means there is no interference to capture or there is an interference $(r, \mathsf{com}) \in \mathbb{I}$ with $o \subseteq r$. We write $\mathbb{I} * r$ for the set of interferences $(o * r, \mathsf{com})$ with $(o, \mathsf{com}) \in \mathbb{I}$. We also use the notation for sets of predicates $\mathbb{P}$ and write $\mathbb{P} * r$ for the set of predicates $p * r$ with $p \in \mathbb{P}$.

The *interference-freedom check* takes as input a set of interferences $\mathbb{I}$ and a set of predicates $\mathbb{P}$. It checks that no interference can invalidate a predicate, $[\![(o, \mathsf{com})]\!](p) \subseteq p$ for all $(o, \mathsf{com}) \in \mathbb{I}$ and all $p \in \mathbb{P}$. If this is the case, we write $\boxplus_{\mathbb{I}} \mathbb{P}$ and say the set of predicates $\mathbb{P}$ is interference-free wrt. $\mathbb{I}$.

The interference-freedom check is non-compositional, and in manual/mechanized program verification this has been the reason to prefer rely-guarantee methods [Feng 2009; Vafeiadis 2008; Vafeiadis and Parkinson 2007]. From the point of view of automated verification, the difference does not matter. After all, there is no compositional way of computing the relies and guarantees.

Program logic. We define the derivation relation $\Vdash$ inductively by the proof rules in Figure 5. COM-SEM only adds the postcondition to the set of predicates to be checked for interference freedom. Similarly, the consequence rule INFER-SEM neither adds the strengthened precondition nor the weakened postcondition. We can freely manipulate predicates as long as there is an interference-free predicate between every pair of consecutive statements, rule SEQ. To ensure this for loops which may be left without execution, rule LOOP adds p to the set of predicates. The initial predicate of the overall Hoare triple is added to the set of predicates by the assumption of Theorem 3 below.

As for the Hoare triples, the proof rules are standard except that they work on the semantic level. This is best seen in rule COM-SEM, which explicitly checks the postcondition for over-approximating the postimage.

$$\text{COM-SEM}\;\frac{[\![\mathsf{com}]\!](p) \subseteq q}{\{q\}, \mathsf{inter}(p, \mathsf{com}) \Vdash \{\, p \,\}\; \mathsf{com}\; \{\, q \,\}} \qquad \text{INFER-SEM}\;\frac{p \subseteq p' \quad \mathbb{P}, \mathbb{I} \Vdash \{\, p' \,\}\; \mathsf{st}\; \{\, q' \,\} \quad q' \subseteq q}{\mathbb{P}, \mathbb{I} \Vdash \{\, p \,\}\; \mathsf{st}\; \{\, q \,\}}$$

$$\text{FRAME}\;\frac{\mathbb{P}, \mathbb{I} \Vdash \{\, p \,\}\; \mathsf{st}\; \{\, q \,\}}{\mathbb{P} * o, \mathbb{I} * o \Vdash \{\, p * o \,\}\; \mathsf{st}\; \{\, q * o \,\}} \qquad \text{SEQ}\;\frac{\mathbb{P}_1, \mathbb{I}_1 \Vdash \{\, p \,\}\; \mathsf{st}_1\; \{\, q \,\} \quad \mathbb{P}_2, \mathbb{I}_2 \Vdash \{\, q \,\}\; \mathsf{st}_2\; \{\, o \,\}}{\{q\} \cup \mathbb{P}_1 \cup \mathbb{P}_2, \mathbb{I}_1 \cup \mathbb{I}_2 \Vdash \{\, p \,\}\; \mathsf{st}_1 ; \mathsf{st}_2\; \{\, o \,\}}$$

$$\text{LOOP}\;\frac{\mathbb{P}, \mathbb{I} \Vdash \{\, p \,\}\; \mathsf{st}\; \{\, p \,\}}{\{p\} \cup \mathbb{P}, \mathbb{I} \Vdash \{\, p \,\}\; \mathsf{st}^*\; \{\, p \,\}} \qquad \text{CHOICE}\;\frac{\mathbb{P}_1, \mathbb{I}_1 \Vdash \{\, p \,\}\; \mathsf{st}_1\; \{\, q \,\} \quad \mathbb{P}_2, \mathbb{I}_2 \Vdash \{\, p \,\}\; \mathsf{st}_2\; \{\, q \,\}}{\mathbb{P}_1 \cup \mathbb{P}_2, \mathbb{I}_1 \cup \mathbb{I}_2 \Vdash \{\, p \,\}\; \mathsf{st}_1 + \mathsf{st}_2\; \{\, q \,\}}$$

Fig. 5. Program logic.

Theorem 3 (Soundness). $\mathbb{P}, \mathbb{I} \Vdash \{\, p \,\}\; \mathsf{st}\; \{\, q \,\}$ *and* $\boxplus_{\mathbb{I}} \mathbb{P}$ *and* $p \in \mathbb{P}$ *imply* $\models \{\, p \,\}\; \mathsf{st}\; \{\, q \,\}$.

See Appendix E for the detailed proof of Theorem 3.

5 REASONING ABOUT KEYSETS USING FLOWS

Recall from §2 that we localize the reasoning about the abstract state $\mathsf{C}(N)$ of the data structure to the contents $\mathsf{C}(x)$ of a single node x using its keyset $\mathsf{KS}(x)$. In this section, we define the keysets as a derived quantity that we can reason about locally in a separation logic. Then, we use this formalism to define the node-local invariant $\mathsf{Node}(x)$ of our running example. This node-level invariant is used by our tool to fully automatically generate a proof of the Harris list (cf. §8).

We derive the keyset of a node x from another quantity $\mathsf{IS}(x)$, the node's *inset*. Intuitively, $k \in \mathsf{IS}(x)$ if a thread searching for k will traverse node x. For the Harris set, we define $\mathsf{IS}(\mathsf{head}) = [-\infty, \infty]$ and for every other node we obtain $\mathsf{IS}(x)$ as the solution of the following fixpoint equation:

$$\mathsf{IS}(x) \;=\; \bigcup\nolimits_{(y,x) \in E} \; \mathsf{IS}(y) \cap (y.\mathsf{key}, \infty] \;\; .$$

Here, the set of edges E is induced by the next pointers in the heap. If we remove those keys k from $\mathsf{IS}(x)$ for which a search leaves x (i.e., if $k > x.\mathsf{key}$ in the Harris set), we obtain $\mathsf{KS}(x)$. These definitions ensure for free that the keysets are disjoint, the first of our keyset invariants. They also generalize to any search structure [Shasha and Goodman 1988].

To express keysets in separation logic, we use the flow framework [Krishna et al. 2018, 2020b]. In this framework, the heap is augmented by associating every node x with a quantity *flow*(x) that is defined as a solution to a fixpoint equation over the heap like that defining the inset above. Assertions describe disjoint fragments of the augmented global heap, similar to classical separation logic. The augmented heap fragments are called *flow graphs*. In addition to tracking the flow of each node, a flow graph also has an associated *interface* consisting of an *inflow* and an *outflow*. The inflow $in(y, x)$ captures the contribution to the flow of x inside the heap fragment via an edge from a heap node y outside the fragment, and conversely for the outflow $out(x, y)$. Flow graphs fg and fg' compose if they are disjoint and their interfaces are compatible (i.e., the composed flow graph $fg * fg'$ has the same flow as the components). The framework then enables local reasoning about the effects of heap updates on flow graphs. In essence, if a local update inside a region fg of a larger flow graph $fg * fg'$ maintains fg's interface, then the flow in fg' does not change. Hence, any property about fg', such as that each of its nodes x satisfies the keyset invariant $\mathsf{C}(x) \subseteq \mathsf{KS}(x)$, can be framed across the update.

Appendix C provides the technical details of the flow framework adapted to the semantic setting of our program logic. For the remainder of the paper, it suffices to know that we instantiate the framework such that a node's inset can be obtained from its flow as described above.

The Harris' set invariant. We represent predicates $p \subseteq \Sigma$ syntactically using separation logic assertions that are for the most part standard. In particular, we use boxed assertions $\boxed{A}$ that are inspired by RGSep [Vafeiadis 2008; Vafeiadis and Parkinson 2007] to mean that A is interpreted in the global state. Unboxed assertions are interpreted in the local state. A *points-to* predicate takes the form $x \mapsto \langle \overline{\mathsf{sel}_i : t_i}, \mathtt{flow} : t_{\mathit{flow}}, \mathtt{in} : t_{\mathit{in}} \rangle$ and describes a flow graph consisting of a single node x. Here, each sel_i is a field selector and t_i is a term denoting the field's associated value. The *ghost field* `flow` stores x's flow and `in` stores its inflow. The semantics of assertions is defined by a satisfaction relation, which induces a denotation function $[\![A]\!] \subseteq \Sigma$. We defer the technical details to Appendix D as they are standard.

We next define the resources associated with a node x, its inset, and keyset. In proofs, we will assume that assertions are existentially closed, and will omit the corresponding outer quantifiers. Formulas like $\mathsf{Node}(x)$ defined in the following introduce logical variables like $\mathit{mark}(x)$ that are visible beyond $\mathsf{Node}(x)$, for example, to define the keyset term $\mathsf{KS}(x)$. We define:

$$\mathsf{Node}(x) \triangleq \boxed{x \mapsto \langle \mathtt{mark}\colon \mathit{mark}(x), \mathtt{next}\colon \mathit{next}(x), \mathtt{key}\colon \mathit{key}(x), \mathtt{flow}\colon \mathit{flow}(x), \mathtt{in}\colon \mathit{in}(x) \rangle}$$

$$\mathsf{IS}(x) \triangleq x = \mathsf{head}\ ?\ [-\infty, \infty]\ :\ \mathit{flow}(x) \qquad\qquad \mathsf{KS}(x) \triangleq \mathsf{IS}(x) \setminus (\mathit{key}(x), \infty]\ .$$

With these definitions in place, we define the resource invariant $\mathsf{Inv}(N', N)$ that is maintained by each subregion $N' \subseteq N$ of a Harris set structure consisting of nodes N:

$$\begin{aligned}
\mathsf{Inv}(N', N) &\triangleq \mathsf{HD}(N) * \mathop{\Huge\ast} x \in N'.\ \mathsf{Node}(x) * \varphi^1(x) * \varphi^2(x) * \varphi^3(x, N) * \varphi^4(x) \\
\mathsf{HD}(N) &\triangleq \mathsf{head} \in N * \mathit{key}(\mathsf{head}) = -\infty * \neg \mathit{mark}(\mathsf{head}) \\
\varphi^1(x) &\triangleq \neg \mathit{mark}(x) \implies \mathsf{IS}(x) \neq \varnothing \\
\varphi^2(x) &\triangleq \mathsf{IS}(x) \neq \varnothing \implies [\mathit{key}(x), \infty) \subseteq \mathsf{IS}(x) \\
\varphi^3(x, N) &\triangleq \{x, \mathit{next}(x)\} \in N \land (\mathit{key}(x) = \infty \implies \neg \mathit{mark}(x)) \\
\varphi^4(x) &\triangleq \forall y, z.\ \mathit{in}(x)(y, x) \neq \varnothing * \mathit{in}(x)(z, x) \neq \varnothing \implies y = z
\end{aligned}$$

Formula $\varphi^1(x)$ captures that all unmarked nodes are reachable from head. Formula $\varphi^2(x)$ implies the second keyset invariant $C(x) \subseteq \mathsf{KS}(x)$ and will also allow us to establish $k \in \mathsf{KS}(x)$ at the appropriate points in the proof. Formula $\varphi^3(x)$ ensures $N' \subseteq N$ and that N is closed under traversal of next pointers. It also implies that the tail node is unmarked. Finally, $\varphi^4(x)$ implies that there exists at most one path from head to each x that a traversal would actually follow. This is needed to prove that unlinking marked nodes from the structure preserves the invariant. It is worth noting that the invariant does not put many constraints on the data structure shape. The sole purpose is to provide enough information to reason about the keysets.

If N is clear, we abbreviate $\mathsf{Inv}(N', N)$ to $\mathsf{Inv}(N')$. We will further freely use the following fact, which states that we can arbitrarily split and merge the invariant for disjoint subregions of the data structure for the purpose of framing.

Lemma 4. $\mathsf{Inv}(N_1 \uplus N_2, N)$ *iff* $\mathsf{Inv}(N_1, N) * \mathsf{Inv}(N_2, N)$.

6 FUTURES

We now make our program logic future-proof. We refer to the set of nodes affected by the execution of a command as the command's *footprint*. A command affects a node x if it changes a field value of x or the flow at x. The footprint of an update on a flow graph is in general larger than the footprint of the same update on the underlying heap graph alone. For instance, $l.\mathtt{next} := r$ does not abort as long as the location at l is in the heap graph. However, if the same command is executed on a flow graph, it will typically require other nodes such as r and $l.\mathtt{next}$ to be present in order for the

$$\textsc{f-intro}\ \frac{p \subseteq wp(\mathsf{com}, q)}{\mathsf{emp} \ \subseteq\ \langle\, p \,\rangle\, \mathsf{com}\, \langle\, q \,\rangle} \qquad \textsc{f-seq}\ \frac{wp(\mathsf{com}_1; \mathsf{com}_2, o) \subseteq wp(\mathsf{com}, o)}{\langle\, p \,\rangle\, \mathsf{com}_1\, \langle\, q \,\rangle * \langle\, q \,\rangle\, \mathsf{com}_2\, \langle\, o \,\rangle \ \subseteq\ \langle\, p \,\rangle\, \mathsf{com}\, \langle\, o \,\rangle}$$

$$\textsc{f-infer}\ \frac{p_2 \subseteq p_1 \qquad q_1 \subseteq q_2}{\langle\, p_1 \,\rangle\, \mathsf{com}\, \langle\, q_1 \,\rangle \ \subseteq\ \langle\, p_2 \,\rangle\, \mathsf{com}\, \langle\, q_2 \,\rangle} \qquad \textsc{f-frame}\ \langle\, p \,\rangle\, \mathsf{com}\, \langle\, q \,\rangle \ \subseteq\ \langle\, p * o \,\rangle\, \mathsf{com}\, \langle\, q * o \,\rangle$$

$$\textsc{f-account}\ p * \langle\, p * q \,\rangle\, \mathsf{com}\, \langle\, o \,\rangle \ \subseteq\ \langle\, q \,\rangle\, \mathsf{com}\, \langle\, o \,\rangle \qquad \textsc{f-invoke}\ p * \langle\, p \,\rangle\, \mathsf{com}\, \langle\, q \,\rangle \ \subseteq\ wp(\mathsf{com}, q)$$

Fig. 6. Implications among futures.

command not to abort. This is because the command may change the flow of these other nodes. For instance, the footprint of the `CAS(l.next, ln, r)` on Line 16 of Figure 1 comprises the entire marked segment between l and r, because it changes the flow and hence the keysets of all nodes in the segment. We introduce futures to reason about commands with such unbounded footprints.

As futures admit general reasoning principles, we study them in the abstract semantic setting of §4 and then apply the developed principles to our concrete running example.

Reasoning about futures. Futures are expressed in terms of weakest preconditions. We define the weakest precondition, $wp(\mathsf{com}, q)$, of a command com and predicate q in the expected way: $wp(\mathsf{com}, q) \triangleq \{\, \mathsf{s} \in \Sigma \mid [\![\mathsf{com}]\!](\mathsf{s}) \subseteq q \,\}$. The weakest precondition of the sequential composition $\mathsf{com}_1; \mathsf{com}_2$ is also defined as usual: $wp(\mathsf{com}_1; \mathsf{com}_2, q) \triangleq wp(\mathsf{com}_1, wp(\mathsf{com}_2, q))$.

Definition 5. *Futures are* $\langle\, p \,\rangle\, \mathsf{com}\, \langle\, q \,\rangle \triangleq p \mathrel{-\!\!*} wp(\mathsf{com}, q)$.

As futures are predicates, our assertion language treats futures as first-class assertions.

Readers familiar with Iris [Jung et al. 2018] will note that our definition of futures resembles Iris' notion of Hoare triples. The key difference is that in Iris, the predicate $p \mathrel{-\!\!*} wp(\mathsf{com}, q)$ is additionally guarded by a persistence modality, making Hoare triples duplicable resources. That is, a Hoare triple in Iris can be used arbitrarily many times. In contrast, futures can carry resources and may therefore be subject to interference. Hence, they are not duplicable *per se*. This is a deliberate design choice motivated by the reasoning technique of accounting. We note that a future $\langle\, p \,\rangle\, \mathsf{com}\, \langle\, q \,\rangle$ is an ordinary duplicable Hoare triple if $\mathsf{emp} \subseteq \langle\, p \,\rangle\, \mathsf{com}\, \langle\, q \,\rangle$.

Figure 6 gives the implications we use for reasoning about futures. The rule F-INTRO turns an ordinary Hoare triple that proves the correctness of a command com into a future. The rules F-INFER and F-FRAME correspond to the rules INFER-SEM and FRAME for Hoare triples. The rule F-INVOKE allows us to invoke a future $\langle\, p \,\rangle\, \mathsf{com}\, \langle\, q \,\rangle$ at the point in the proof where the update chunk com is actually executed. That is, we can use this rule to discharge the premise of rule COM-SEM.

The composition of ghost update chunks is implemented by rule F-SEQ. It is similar to the rule for sequential composition in Hoare logic with two important differences. First, it requires a separating conjunction of the futures for the composed update chunks com_1 and com_2. The reason is that futures may carry resources. Second, it replaces the composition of com_1 and com_2 by a new update chunk com that is equivalent. Unlike the rule SEQ, the rule F-SEQ does not take into account interferences on the intermediate assertion p. This is correct, since update chunks represent ghost computation that takes effect instantaneously, meaning com_1 and com_2 are executed uninterruptedly at the moment when the new update chunk com is invoked.

Finally, the rule F-ACCOUNT enables the partial invocation of a future $\langle\, p * q \,\rangle\, \mathsf{com}\, \langle\, o \,\rangle$ by eliminating the premise p if it is present in the current proof context. We refer to this rule as *accounting*. We have

$\mathsf{TInv}(N, M, l, ln, lmark, t) \triangleq \mathsf{Inv}(N) * \neg lmark * key(l) < k < \infty * \{\, l, ln, t \,\} \subseteq M \subseteq N$
$\mathsf{Fut}(M, l, ln, t) \triangleq \langle\, \mathsf{P}(M, l, ln, t, ln) \,\rangle\; l.\mathsf{next} := t \;\langle\, \mathsf{Q}(M, l, ln, t, t) \,\rangle$
$\mathsf{P}(M, l, ln, t, u) \triangleq \mathsf{Inv}(M) * \{\, l, t \,\} \subseteq M * next(l) = u * \neg mark(l)$
$\mathsf{Q}(M, l, ln, t, u) \triangleq \mathsf{Inv}(M) * \{\, l, t \,\} \subseteq M * next(l) = u * \neg mark(l) * (key(l), \infty] \subseteq flow(t)$

$\{\, \exists N\, M.\ \mathsf{TInv}(N, M, l, ln, lmark, t) * \mathsf{Fut}(M, l, ln, t) \,\}$
procedure `traverse(`k`: K, `l`: N, `ln`: N, `$lmark$`: Bool, `t`: N) {`
  val tn, $tmark$ = **atomic** `{`t`.next, `t`.mark}`
  $\{\, \mathsf{TInv}(N, M, l, ln, lmark, t) * \mathsf{Fut}(M, l, ln, t) * tn \in N * (tmark \Rightarrow mark(t) = tmark * next(t) = tn) \,\}$
  if `(`$tmark$`) {`
    $\{\, \mathsf{TInv}(N, M, l, ln, lmark, t) * \mathsf{Fut}(M, l, ln, t) * tn \in N * mark(t) * mark(t) = tmark * next(t) = tn \,\}$
    $\{\, \mathsf{TInv}(N, M, l, ln, lmark, tn) * \mathsf{Fut}(M, l, ln, tn) \,\}$
    return `traverse(`k`, `l`, `ln`, `tn`)`
  `} ` **else if** `(`t`.key < `k`) {`
    $\{\, \mathsf{TInv}(N, M, l, ln, lmark, tn) * \mathsf{Fut}(M, l, ln, tn) * key(t) < k \,\}$
    return `traverse(`k`, `t`, `tn`, `$tmark$`, `tn`)`
  `} ` **else** `{`
    $\{\, \mathsf{TInv}(N, M, l, ln, lmark, t) * \mathsf{Fut}(M, l, ln, t) * t \neq \mathsf{head} * t \neq l * key(l) < k \leq key(t) \,\}$
    return `(`l`, `ln`, `$lmark$`, `t`)`
`} }`
$\{\, (l, ln, lmark, r).\ \exists N\, M.\ \mathsf{TInv}(N, M, l, ln, lmark, r) * \mathsf{Fut}(M, l, ln, r) * r \neq \mathsf{head} * r \neq l * key(l) < k \leq key(r) \,\}$

Fig. 7. Proof outline showing that Harris' set `traverse` prepares the `CAS` from `search`. The preparation guarantees that the `CAS` *will* maintain the invariant. We capture this with a future.

already seen in §2 that accounting is useful to enable the composition of two futures using F-SEQ. Note that if p is subject to inference, so is the future $\langle\, q \,\rangle\ \mathsf{com}\ \langle\, o \,\rangle$ obtained from rule F-ACCOUNT.

The following lemma states the soundness of the rules.

Lemma 6. *The rules* F-INTRO, F-SEQ, F-INFER, F-FRAME, F-ACCOUNT, *and* F-INVOKE *are valid implications.*

Proving the Harris set invariant. We demonstrate the versatility of futures by using them to prove that the `CAS` on line 16 of the Harris set preserves the invariant of the data structure. Figure 7 shows the proof outline. The code is equivalent to the one in Figure 1, except that the mark bits have been made explicit. We discuss the key aspects of the proof in more detail.

The precondition of `traverse` contains the predicate $\mathsf{TInv}(N, M, l, ln, lmark, t)$. It is the invariant of `traverse` and states that the data structure's invariant $\mathsf{Inv}(N)$ is maintained. Additionally, the precondition contains the future $\mathsf{Fut}(M, l, ln, t)$. It captures the fact that the segment from l to t consisting of the nodes $M \setminus \{\, l, t \,\}$ can be safely unlinked via the update $l.\mathtt{next} := t$, provided $l.\mathtt{next} = ln$ and $\neg mark(l)$. After the update, the future guarantees that we have $(key(l), \infty] \subseteq flow(t)$, a crucial fact we will later use in the linearizability proof (cf. §7).

To satisfy the precondition when invoking `traverse` from `find`, observe that the invocation is of the form `traverse(`k`, head,`hn, hn`)` where hn stems from hn `= head.next`. The invariant $\mathsf{TInv}(N, \{\mathsf{head}, hn\}, \mathsf{head}, hn, hmark, hn)$ of `traverse` follows from the data structure invariant $\mathsf{Inv}(N)$. The future $\mathsf{Fut}(M, \mathsf{head}, hn, hn)$ can be obtained trivially via F-INTRO because the update $\mathsf{head.next} := hn$ has no effect if $next(\mathsf{head}) = hn$.

The postcondition of `traverse` contains the invariant $\mathsf{TInv}(N, M, l, ln, lmark, r)$ and the future $\mathsf{Fut}(M, l, ln, r)$. By applying rule F-INVOKE, we can use the future to prove the correctness of the case where the `CAS(`l`.next,`ln, r`)` at Line 16 (Figure 1) succeeds. The remaining facts of the postcondition state that `traverse` has found the part of the data structure that contains the search key k if present.

The most interesting part of the proof is the transition between lines 34 and 35, particularly the transition from $\mathsf{Fut}(M, l, ln, t)$ to $\mathsf{Fut}(M, l, ln, tn)$. Here, we need to extend the marked segment M by adding tn. This step involves an application of F-SEQ to compose the update chunk for $l.\mathtt{next} := t$ with the one for $l.\mathtt{next} := tn$. We elaborate this step in detail, extending the discussion in §2.

We start from $\mathsf{Fut}(M, l, ln, t)$ and use rule F-FRAME to extend both sides of this future with $\mathsf{Inv}(tn)$. The resulting future can be rewritten into the form

$$\left\langle\, \hat{\mathsf{P}}(l, t, tn, ln) \;*\; \mathsf{Inv}(M \setminus \{\, l, t, tn \,\}) \,\right\rangle \; l.\mathtt{next} := t \; \left\langle\, \hat{\mathsf{Q}}(l, t, tn, t) \;*\; \mathsf{Inv}(M \setminus \{\, l, t, tn \,\}) \,\right\rangle$$

$$\begin{aligned} \text{where} \quad \hat{\mathsf{P}}(l, t, tn, u) \;&\triangleq\; \mathsf{Inv}(\{\, l, t, tn \,\}) \;*\; next(l) = u \;*\; \neg mark(l) \\ \hat{\mathsf{Q}}(l, t, tn, u) \;&\triangleq\; \mathsf{Inv}(\{\, l, t, tn \,\}) \;*\; next(l) = u \;*\; \neg mark(l) \;*\; (key(l), \infty] \subseteq flow(t)\;. \end{aligned}$$

This future plays the role of $\langle\, p \,\rangle \; \mathtt{com}_1 \; \langle\, q \,\rangle$ in our application of rule F-SEQ. To obtain the future playing the role of $\langle\, q \,\rangle \; \mathtt{com}_2 \; \langle\, o \,\rangle$, we proceed in multiple steps. First, we use F-INTRO to derive

$$\left\langle\, \hat{\mathsf{P}}(l, t, tn, t) \;*\; next(t) = tn * mark(t) \,\right\rangle \; l.\mathtt{next} := tn \; \left\langle\, \hat{\mathsf{Q}}(l, t, tn, tn) \,\right\rangle \;.$$

To satisfy the premise of F-INTRO, we need to show that (i) the update $l.\mathtt{next} := tn$ is frame-preserving, i.e., the interface of the flow graph consisting of nodes $\{l, t, tn\}$ does not change, and (ii) the invariants $\varphi^i(x)$ are preserved for all $x \in \{l, t, tn\}$. First observe that $\neg mark(l)$ and $\varphi^1(l)$ imply $\mathsf{IS}(l) \neq \emptyset$. Using $key(l) < \infty$, $\varphi^4(t)$, and the fixpoint equation defining insets, we obtain $\mathsf{IS}(t) = \mathsf{IS}(l) \cap (key(l), \infty] \neq \emptyset$. From $\varphi^3(t)$ and $mark(t)$ we obtain $key(t) \neq \infty$. Thus, using similar reasoning as above, we conclude $\mathsf{IS}(tn) = \mathsf{IS}(t) \cap (key(t), \infty] \neq \emptyset$. The inset and inflow of l are unaffected by the update, so its invariant is trivially preserved. For t, let $\mathsf{IS}'(t) = \emptyset$ denote the new inset. Since t is marked, this means that all its invariants are preserved. The new inset of tn is $\mathsf{IS}'(tn) = \mathsf{IS}(l) \cap (key(l), \infty]$. Observe that we have $\mathsf{IS}(tn) \subseteq \mathsf{IS}'(tn) \neq \emptyset$, so the invariants for tn are also maintained. Finally, to show that the interface of the modified region remains the same, it suffices to prove $\mathsf{IS}(tn) \cap (key(tn), \infty] = \mathsf{IS}'(tn) \cap (key(tn), \infty]$. This holds true if $key(t) < key(tn)$, which follows from $\varphi^2(tn)$ and $\emptyset \neq \mathsf{IS}(tn) \subseteq (key(t), \infty]$.

Next, we apply F-ACCOUNT for $next(t) = tn * mark(t)$ from the proof context and use F-FRAME to add the remaining part $\mathsf{Inv}(M \setminus \{\, l, t, tn \,\})$ of the segment M as a frame. This yields

$$\left\langle\, \hat{\mathsf{P}}(l, t, tn, t) \;*\; \mathsf{Inv}(M \setminus \{\, l, t, tn \,\}) \,\right\rangle \; l.\mathtt{next} := tn \; \left\langle\, \hat{\mathsf{Q}}(l, t, tn, tn) \;*\; \mathsf{Inv}(M \setminus \{\, l, t, tn \,\}) \,\right\rangle \;.$$

We can now use F-SEQ with this future and the one derived above. Note that the premise of the rule follows easily because $\mathtt{com}_1$ and $\mathtt{com}_2$ update the same memory location and $\mathtt{com}_2 = \mathtt{com}$. We obtain

$$\left\langle\, \hat{\mathsf{P}}(l, t, tn, ln) \;*\; \mathsf{Inv}(M \setminus \{\, l, t, tn \,\}) \,\right\rangle \; l.\mathtt{next} := tn \; \left\langle\, \hat{\mathsf{Q}}(l, t, tn, tn) \;*\; \mathsf{Inv}(M \setminus \{\, l, t, tn \,\}) \,\right\rangle \;.$$

Applying F-INFER and introducing a fresh existential M', we rewrite this into the form

$$M' = M \cup \{\, l, t, tn \,\} \;*\; \left\langle\, \mathsf{P}(M', l, ln, tn, ln) * t \in M' \,\right\rangle \; l.\mathtt{next} := tn \; \left\langle\, \mathsf{Q}(M', l, ln, tn, tn) \,\right\rangle.$$

Now, we apply F-ACCOUNT one more time for $t \in M'$ and use $tn \in N$, $ln \in M$, and $M \subseteq N$ from the proof context, to obtain

$$M' \subseteq N \;*\; \{\, l, ln, tn \,\} \subseteq M' \;*\; \mathsf{Fut}(M', l, ln, tn) \;\;.$$

This allows us to reestablish $\mathsf{TInv}(N, M', l, ln, lmark, tn) * \mathsf{Fut}(M', l, ln, tn)$. As M' and M are existentially quantified, we can finally rename M' to M, which yields the assertion on Line 35.

Checking interference freedom. We briefly discuss why the proof is interference free relative to other threads performing set operations. First, all commands maintain $\mathsf{Inv}(N)$ and N can only grow larger. Next, assertions depending on field `key` are interference free since a node's `key` is never changed after initialization. Similarly, `mark` is only changed monotonically from *false* to *true*. Moreover, `next` is only changed for unmarked nodes (e.g., the proof guarantees $\neg mark(l)$ in the successful case of the CAS on Line 54 and the insert operation provides a similar guarantee). This is why assertions such as $mark(t)$ on Line 34 are interference free. Finally, futures constructed using rule F-INTRO are always interference free. All remaining futures are constructed by accounting interference-free facts or by composing interference-free futures via F-SEQ.

7 HISTORIES

We next present an extension of our developed theory that allows us to reason about *separated computation histories*. We integrate a form of hindsight reasoning for propagating knowledge between current and past states—hindsight is a key technique to handle non-fixed linearization points [Feldman et al. 2018, 2020; Lev-Ari et al. 2015; O'Hearn et al. 2010]. We develop the new theory again in the general setting of §3 and §4 and then apply it to our running example.

History separation algebras. Recall that our states are taken from a separation algebra $(\Sigma, *, \mathsf{emp})$. We refer to a non-empty sequence of states, $\sigma \in \Sigma^+$, as a *computation history*. Computation histories also form a separation algebra where composition on states is lifted to computation histories as follows. First, for sequences $\sigma = \mathsf{s}_1 \dots \mathsf{s}_n$ and $\tau = \mathsf{t}_1 \dots \mathsf{t}_m$, the composition $\sigma * \tau$ is defined, written $\sigma \# \tau$, iff $n = m$ and for all i, $\mathsf{s}_i \# \mathsf{t}_i$. In this case, we let $\sigma * \tau \triangleq (\mathsf{s}_1 * \mathsf{t}_1) \dots (\mathsf{s}_n * \mathsf{t}_n)$. The set of units is given by emp^+.

Lemma 7. $(\Sigma^+, *, \mathsf{emp}^+)$ *is a separation algebra.*

Predicates $a, b, c \subseteq \Sigma^+$ now refer to sets of computations. We lift the semantics of commands to computation predicates in the expected way.

Definition 8. $[\![\mathsf{com}]\!](\sigma.\mathsf{s}_1) \triangleq \{\, \sigma.\mathsf{s}_1.\mathsf{s}_2 \mid \mathsf{s}_2 \in [\![\mathsf{com}]\!](\mathsf{s}_1) \,\}$.

However, the locality assumption (LocCom) on the semantics of commands that is needed for the soundness of framing does not necessarily carry over from state predicates to computation predicates. If we want to frame computation predicates, we have to make an additional assumption.

Definition 9. *A predicate $a \subseteq \Sigma^+$ is* frameable*, if it satisfies* $\forall \sigma. \forall \mathsf{s}.\ \sigma.\mathsf{s} \in a \implies \sigma.\mathsf{s}.\mathsf{s} \in a$.

Lemma 10. *If c is frameable,* $[\![\mathsf{com}]\!](a * c) \subseteq [\![\mathsf{com}]\!](a) * c$.

We lift the semantics of concurrency libraries to history separation algebras $(\Sigma_{\mathrm{G}} \times \Sigma_{\mathrm{L}})^+$. The notions of initial and accepting configurations, and soundness remain unchanged except that they now range over computation predicates instead of state predicates. The technical details of this lifting are straightforward, we defer them to Appendix G. The soundness guarantee in Theorem 3 continues to hold for history separation algebras modulo a subtlety. We can only apply rule FRAME if the predicate to be added is frameable in the sense of Definition 9.

Theorem 11 (Soundness). $\mathbb{P}, \mathbb{I} \Vdash \{\, a \,\}\ \mathsf{st}\ \{\, b \,\}$ *and* $\boxtimes_{\mathbb{I}} \mathbb{P}$ *and* $a \in \mathbb{P}$ *imply* $\models \{\, a \,\}\ \mathsf{st}\ \{\, b \,\}$.

Frameable computation predicates. We next discuss general principles for constructing frameable computation predicates from state predicates.

Definition 12. *State predicate $p, q \subseteq \Sigma$ yield the following predicates over computations:*
(i) The now predicate $\_p \ \triangleq\ \Sigma^*.p$.
(ii) The past predicate $\diamondsuit p \ \triangleq\ \Sigma^*.p.\Sigma^*$.

The now predicate refers to the current state. The past predicate allows us to track auxiliary information about the computation. These predicates work well in our setting in that they are frameable.

Lemma 13. *(i) $\_p$ and $\diamondsuit p$ are frameable. (ii) If a and b are frameable, so are $a * b$, $a \cap b$, and $a \cup b$.*

Frameability is not preserved under complementation and separating implication. However, the now operator is compatible with the SL operators in a strong sense.

Lemma 14. *$\_(p \oplus q) = \_p \oplus \_q$ for all $\oplus \in \{\cap, \cup, *, \mathbin{-\!\!*}\}$, $\_(\overline{p}) = \overline{\_p}$, $\mathit{false} = \_\mathit{false}$, $\mathit{true} = \_\mathit{true}$, and $\_p \subseteq \_q$ iff $p \subseteq q$.*

For the past operator, we rely on the properties stated by the following lemma. In particular, the last equivalence justifies the rule H-INFER used in 2.

Lemma 15. *$\_p \subseteq \diamonddot p$, $\mathit{true} = \diamonddot \mathit{true}$, $\mathit{true} * \diamonddot p = \diamonddot(p * \mathit{true})$, $\mathit{false} = \diamonddot \mathit{false}$, $\diamonddot(p * q) \subseteq \diamonddot p * \diamonddot q$, $\diamonddot p \mathbin{-\!\!*} \diamonddot q \subseteq \diamonddot(p \mathbin{-\!\!*} q)$, $\diamonddot(p \cap q) \subseteq \diamonddot p \cap \diamonddot q$, $\diamonddot(p \cup q) = \diamonddot p \cup \diamonddot q$, and $\diamonddot p \subseteq \diamonddot q$ iff $p \subseteq q$.*

The interplay between computation predicates and commands is stated in the following lemma. Recall that we defined $wp(\mathtt{com}, a) \triangleq \{\, \sigma \mid [\![\mathtt{com}]\!](\sigma) \subseteq a \,\}$.

Lemma 16. *We have (i) $wp(\mathtt{com}, \_p) = \_wp(\mathtt{com}, p)$, and (ii) $wp(\mathtt{com}, \diamonddot p) = \diamonddot p \cup wp(\mathtt{com}, \_p)$.*

The first identity of Lemma 16 implies that interference checking for a now predicate reduces to inference checking for the underlying state predicate. The second implies that past predicates are interference-free for all commands.

Next we justify the rule H-INTRO used in §2. Recall that this rule provides a way to record information about the current state in a past predicate so that we can use this information later in the proof. This involves a stuttering step.

Lemma 17. *$\_p \subseteq wp(\mathtt{skip}, \_p * \diamonddot p)$.*

Hindsight Reasoning. We now use history separation algebras to justify the hindsight reasoning principle introduced in §2. The key idea is *state-independent quantification*, and best explained with reference to an assertion language. An assertion language over computations will support quantified logical variables. As those quantifiers live on the level of computations, the resulting valuation of the logical variables will be independent of (the same for all) the states inside the computation. This means facts that we learn about the variables in one state will also be true in all other states. In particular, if we learn facts about a quantified variable now, we can draw conclusions in hindsight. We illustrate this on an example. In the assertion $\exists v.(\diamonddot(x \mapsto v) * \_(v = 0))$, the logical variable v is quantified on the level of computations, meaning its value is independent of the actual state in the computation. We learn that now v is zero, and since the valuation is state independent, v has also been zero when x pointed to it. Hence, from the now state we can conclude, in hindsight, that also $\diamonddot(x \mapsto 0)$ holds. This is indeed a consequence of the previous assertion (entailment holds). Rather than moving to an assertion language, we formalize this reasoning on the semantic level.

For hindsight reasoning, we construct a product separation algebra $\Sigma^+ \times I$. The first component is the above history separation algebra $(\Sigma^+, *, \mathsf{emp}^+)$. We refer to the second component as *valuation* or *stack* separation algebra $(I, *, I)$, because its elements can be understood as variable valuations. There are no requirements on I, it is just an arbitrary set, but we note that it is also the set of units. The multiplication $*$ between valuations $i, j \in I$ is defined if and only if $i = j$, in which case $i * i = i$. The product separation algebra $\Sigma^+ \times I$ is the verbatim translation of the idea of state-independent quantification into the semantic world.

Predicates $a \subseteq \Sigma^+ \times I$ over the product separation algebra are sets of pairs (σ, i) consisting of a computation and a valuation. We lift the predicates $\_p$ and $\diamonddot p$ to $\Sigma^+ \times I$ and let them refer to sets of the form $\_p \times J$ resp. $\diamonddot p \times J$ with $J \subseteq I$. So, strictly speaking, the now and the past operator are relations rather than functions in this setting. Separation logic assertions are called pure, if they are independent of the heap and only refer to the valuation of logical variables. In the semantic setting, we define a predicate a to be *pure*, if it belongs to the set of units in the product separation algebra, $a \subseteq \mathsf{emp}^+ \times I$. The following lemma then justifies the rule H-HINDSIGHT used in §2.

```
45 Past(r) ≜ key(r) = ‘key(r) ∗ (‘mark(r) ⇒ mark(r)) ∗ ⟐(‘Node(r) ∗ ‘(k ∈ KS(r)))
46 { ∃N. Inv(N) ∗ −∞ < k < ∞ }
47 procedure find(k: K) : N * N {
48   val hn, hmark = atomic {head.next, head.mark}
49   { TInv(N, {head, hn}, head, hn, hmark, hn) ∗ Fut({head, hn}, head, hn, hn) }
50   val l, ln, lmark, r = traverse(k, head, hn, hmark, hn)
51   { TInv(N, M, l, ln, lmark, r) ∗ Fut(M, l, ln, r) ∗ r ≠ head ∗ t ≠ l ∗ key(l) < k ≤ key(r) }
52   val succ = atomic { // CAS
53     l.next == ln && l.mark == lmark ? {
54       { next(l) = ln ∗ ¬mark(l) ∗ { l, ln, t } ⊆ M ⊆ N ∗ Inv(N) ∗ Fut(M, l, ln, r) ∗ key(l) < k ≤ key(r) }
55       l.next := r
56       { Inv(N) ∗ { l, r } ⊆ N ∗ (key(l), ∞] ⊆ flow(r)
           ∗ key(l) < k ≤ key(r) }
57       skip
58       { Inv(N) ∗ { l, r } ⊆ N ∗ (key(l), ∞] ⊆ flow(r)
           ∗ key(l) < k ≤ key(r) ∗ Past(r) }
59     } true : false
60   }
61   { Inv(N) ∗ { l, r } ⊆ N ∗ (succ ⇒ Past(r)) }
62   if (succ && !r.mark) {
63     { Inv(N) ∗ { l, r } ⊆ N ∗ ¬‘mark(r) ∗ Past(r) }
64     return (l, r)
65   } else find(k)
66 }
67 { (l, r). ∃N. Inv(N) ∗ { l, r } ⊆ N ∗ ¬‘mark(r) ∗ Past(r) }
```

```
68 { ∃N. Inv(N) ∗ −∞ < k < ∞
       ∗ Obl(search(k)) }
69 procedure search(k: K) : Bool {
70   val _, r = find(k)
71   { Inv(N) ∗ r ∈ N ∗ ¬‘mark(r)
       ∗ Past(r) ∗ Obl(search(k)) }
72   val res = r.key == k
73   { Inv(N) ∗ r ∈ N ∗ Obl(search(k))
       ∗ ⟐(‘Node(r) ∗ res ⇔ k ∈ ‘C(N)) }
74   { Inv(N) ∗ Ful(search(k), res) }
75   return res
76 }
77 { res. Inv(N) ∗ Ful(search(k), res) }
```

Fig. 8. Proof outline showing that Harris' set `search` preserves the data structure invariant and is linearizable.

Lemma 18. *If $\_p$ is pure, then $\_p * \diamond q \;\subseteq\; \diamond(p * q)$.*

Proving linearizability of the Harris set. By using the now predicate, we obtain a conservative extension of separation logic. In the following, the application of the now operator is kept implicit: a state predicate that occurs in a context expecting a computation predicate is interpreted as a now predicate. This is justified by Lemma 14.

We demonstrate the reasoning power of the resulting logic by proving linearizability of the `search` operation of the Harris set. The proof outline is in Figure 8, reusing our earlier proof of `traverse`. The code of `find` makes the semantics of the CAS explicit. We have also eliminated the case $ln == r$ before the CAS. This focuses the discussion on a single linearization point. Appendix A gives the proof for the full version.

As in §2, we decorate logical variables occurring below a past operator with a prime to avoid clashes with variables describing the current state.

We proceed with the proof. First, observe the history assertion in the postcondition of `find`

$$\diamond\big(`\mathrm{Node}(r) * `(k \in \mathrm{KS}(r))\big) \;*\; \neg`mark(r) \;*\; key(r) = `key(r)\;.$$

Following our discussion from §2, the assertion $`\mathrm{Node}(r) * `(k \in \mathrm{KS}(r))$ below the past operator together with $r \in N * \neg`mark(r) * key(r) = `key(r)$ allows us to apply the keyset reasoning from §5: we obtain the predicate $\diamond\big(k \in `\mathrm{C}(N) \Leftrightarrow k = key(r)\big)$, Line 73. This means that `search` is linearizable.

To guarantee that the operation linearizes exactly once during its execution, we track an auxiliary ghost resource in the proof [Vafeiadis 2008]. We write $\mathsf{Obl}(\texttt{search}(k))$ to denote the obligation to linearize operation `search` for key k, Line 68. If a linearization point is found, we turn the obligation into a fulfillment $\mathsf{Ful}(\texttt{search}(k), b)$ where b is a Boolean indicating the expected return value. Here, we support hindsight reasoning as well: we let the above past predicate fulfill the obligation to linearize. More concretely, we use the following meta rules to discharge linearizability obligations for pure linearization points, i.e., those that do not modify the abstract state $\mathsf{C}(N)$:

$$\mathsf{Obl}(\texttt{search}(k)) * \diamondsuit(k \in \mathsf{C}(N)) \subseteq \mathsf{Ful}(\texttt{search}(k), \mathit{true})$$
$$\text{and} \qquad \mathsf{Obl}(\texttt{search}(k)) * \diamondsuit(k \notin \mathsf{C}(N)) \subseteq \mathsf{Ful}(\texttt{search}(k), \mathit{false}) \;.$$

There are similar rules for the pure cases of `insert` and `delete`. For impure linearization points, obligations are turned into fulfillments in the current state, at the moment the linearization point occurs. We do not go into the details of these ghost resources, they are standard and orthogonal.

To prove the postcondition of `find`, we need to establish that at some point during the execution of `find`, $k \in \mathsf{KS}(r)$ and $\neg \mathit{mark}(r)$ were satisfied. This is on Line 56, but we determine r's mark bit only later as execution continues past the condition on Line 62. The high-level proof idea is, thus, to record r's keyset and mark bit on Line 56 in the past assertion shown on Line 58 (we explain this step in more detail below). We then propagate the assertion on Line 58 into the *then* branch of the conditional (using Lemma 16). The logical variables link the past and current state, which allows us to apply hindsight reasoning (Rule H-HINDSIGHT). Specifically, on Line 63 we know that $\mathit{mark}(r)$ is false in the current state. Then, using $`\mathit{mark}(r) \Rightarrow \mathit{mark}(r)$, we can conclude that $`\mathit{mark}(r)$ is also false, thus learning retrospectively the crucial fact about the past state at Line 56. We then transfer this pure fact inside the past predicate to derive the desired postcondition on Line 63.

We briefly discuss the mechanical aspects of deriving the past predicate on Line 58. We start with the intermediate assertion established on Line 56 in the earlier proof. First, using $\mathit{key}(l) < k$ and $(\mathit{key}(l), \infty] \subseteq \mathit{flow}(r)$ we derive $k \in \mathsf{IS}(r)$. Then, using $k \leq \mathit{key}(r)$ we obtain $k \in \mathsf{KS}(r)$. Next, we use $r \in N$ to extract $\mathsf{Node}(r)$ from $\mathsf{Inv}(N)$ and derive $\mathsf{Node}(r) * k \in \mathsf{KS}(r)$. We then perform a stuttering step to record a copy of this assertion below a past operator using Lemma 17. After the stuttering step, the original assertion is recombined with the framed remainder of $\mathsf{Inv}(N)$ to obtain:

$$\mathsf{Inv}(N) * \{\, l, r \,\} \subseteq N * \diamondsuit\big(\mathsf{Node}(r) * k \in \mathsf{KS}(r)\big) \;.$$

This assertion is not interference-free since the current state of r referred to inside the past assertion can be changed by concurrent threads. So we perform a series of weakening steps by introducing fresh logical variables to arrive at the assertion on Line 58.

We note that throughout the entire proof, all explicit inductive reasoning was carried out at the level of the program logic in lock-step with the program execution, using only local facts about the nodes in the data structure captured by the resource invariant, futures, and history predicates. In particular, we did not need explicit inductive reasoning over heap graph predicates or computation histories. All such reasoning is carried out *for free* by our developed meta-theory.

8 PROTOTYPE IMPLEMENTATION

We substantiate our claim that the presented techniques aid automation and are useful in practice. To that end, we implemented a `C++` prototype called `plankton`. `plankton` takes as input the program under scrutiny and a candidate node invariant. It then fully automatically generates a proof within our novel program logic, establishing that the given program is linearizable and does adhere to the given invariant. We give a brief overview of `plankton`'s proof generation and report on our findings. We stress that the present paper focuses on the theoretical foundations and as such does not give a detailed discussion of `plankton`'s implementation.

Implementation. The proof generation in `plankton` is implemented as a fixpoint computation that saturates an increasing sequence $\mathbb{I}_0 \subseteq \mathbb{I}_1 \subseteq \cdots$ of interference sets [Henzinger et al. 2003]. Initially, the interference set is empty, $\mathbb{I}_0 \triangleq \varnothing$. Once $\mathbb{I}_k$ is obtained, a proof of the input program with respect to $\mathbb{I}_k$ is constructed and the interferences $\mathbb{I}_{new}$ discovered during this proof are recorded, yielding $\mathbb{I}_{k+1} \triangleq \mathbb{I}_k \cup \mathbb{I}_{new}$. A fixpoint $\mathbb{I}_{lfp} \triangleq \mathbb{I}_k$ is reached if no new interference is found, $\mathbb{I}_k = \mathbb{I}_{k+1}$. The proof generated for $\mathbb{I}_k$ is then the overall proof for the input program.

For the application of interferences during the proof to be efficient, it is crucial to reduce the size of the computed interference sets [Vafeiadis 2010b]. We reduce an interference set $\mathbb{I}$ by dropping any interference $(o, \mathtt{com})$ from $\mathbb{I}$ that is already covered, that is, for which there is $(r, \mathtt{com}) \in \mathbb{I}$ with $o \subseteq r$.

Given an interference set $\mathbb{I}$, `plankton` constructs a proof $\mathbb{I} \Vdash \{\, p_0 \,\}\ \mathtt{fun}\ \{\, q \,\}$ for each function `fun` of the input program. Proof construction starts from the precondition p_0, which captures just the invariant, as done, e.g., in Figure 8. From there, the rules of the program logic (Figure 5) are applied to inductively construct the postcondition. As $\mathbb{I}$ is fixed, plankton does not track the predicates $\mathbb{P}$. We elaborate on the interesting ingredients of the proof construction.

Rule com-sem for atomic commands `com` requires `plankton` to compute $[\![\mathtt{com}]\!](p)$ for some precondition p. The behavior of $[\![\mathtt{com}]\!](p)$ is prescribe by the standard axioms of separation logic [O'Hearn et al. 2001]. If `com` updates the heap, however, we have to additionally infer a flow footprint such that, roughly (i) all nodes updated by `com` are contained in the footprint, (ii) the interface of the footprint remains unchanged by the update, and (iii) after the update all nodes inside the footprint still satisfy their node-local invariant. As discussed in §5, this localizes the update to the footprint: the nodes outside the footprint continue to satisfy the invariant. `plankton` chooses the footprint by collecting all nodes whose (non-flow) fields are updated by `com` and adds those nodes that are reachable in a small, constant number of steps. If this choice does not satisfy (i), verification fails. The restriction to finite footprints is essential for automating (ii) and (iii). Yet, the restriction does not limit our approach: unbounded footprints are handled with futures, as seen in §6. Conditions (ii) and (iii) are then encoded into SMT and discharged using `Z3` [de Moura and Bjørner 2008]. Lastly, we apply the interferences $\mathbb{I}$ to $[\![\mathtt{com}]\!](p)$. The result is $q \triangleq [\![\mathbb{I}]\!]([\![\mathtt{com}]\!](p))$ whose computation is inspired by [Vafeiadis 2010b]. Overall, we obtain $\mathbb{I} \Vdash \{\, p \,\}\ \mathtt{com}\ \{\, q \,\}$.

Rule loop requires a loop invariant I for program $\mathtt{st}^*$ and precondition p such that $\mathbb{I} \Vdash \{\, I \,\}\ \mathtt{st}\ \{\, I \,\}$ and $p \subseteq I$. To find one, `plankton` generates a sequence $I_0, I_1, \ldots$ of candidates. The first candidate is $I_0 \triangleq p$. Candidate I_{n+1} is obtained from a sub-proof $\mathbb{I} \Vdash \{\, I_n \,\}\ \mathtt{st}\ \{\, I'_n \,\}$ whose pre- and postcondition are joined, i.e., $I_{n+1} \triangleq I_n \sqcup I'_n$. Intuitively, this join corresponds to the disjunction $I_n \cup I'_n$. For performance reasons, however, `plankton` uses a disjunction-free domain [Rival and Mauborgne 2007; Yang et al. 2008], which means the join is actually weaker than union. A loop invariant $I \triangleq I_n$ is found, if the implication $I_{n+1} \subseteq I_n$ holds.

A core aspect of our novel program logic are history and future predicates. `plankton` tries to construct a strongest proof for the input program. Hence, new history and future predicates are added eagerly to an assertion p whenever it participates in a join or interference is applied to it. The rational behind this strategy is to *save* information from p in a history/future before it is lost. More specifically, all boxed points-to predicates from p that are subject to interference are added to a new history predicate. New futures are introduced either from scratch with rule f-intro followed by rule f-account or from existing futures with rules f-seq and f-frame. It is worth pointing out that `plankton` uses rule f-account only to account duplicable facts, as in the proof from Figure 7. The introduction of futures is guided by a set of candidates. These candidates are computed upfront by collecting all `CAS` commands in the input program. A `CAS` may be dropped from the candidates if its footprint is statically known to be finite, e.g., because it only updates the mark bit of a pointer or inserts a new (and thus owned) node. The approach discovers the necessary futures needed for our experiments. Avoiding unnecessary futures produced by this method is considered future work.

Table 1. Experimental results for verifying set implementations with `plankton`. The experiments were conducted on an Apple M1 Pro with 16GB of RAM running Ubuntu 20.10 with `clang` 12 and `Z3` 4.8.15.

Benchmark	#Iter	#$\mathbb{I}_{lfp}$	#Cand	Com.	Fut.	Hist.	Join	Inter.	Lineariz.
Fine-Grained set	2	5	2	11%	15%	43%	15%	8%	46*s* ✓
Lazy set	2	6	2	10%	13%	54%	11%	5%	77*s* ✓
FEMRS tree (no maintenance)	2	5	2	19%	0%	49%	1%	9%	130*s* ✓
Vechev&Yahav 2CAS set	2	3	1	14%	0%	33%	31%	9%	125*s* ✓
Vechev&Yahav CAS set	2	4	1	15%	7%	39%	23%	6%	54*s* ✓
ORVYY set	2	3	0	17%	0%	40%	26%	6%	47*s* ✓
Michael set	2	4	2	11%	29%	30%	15%	6%	306*s* ✓
Michael set (wait-free search)	2	4	2	11%	28%	30%	15%	6%	246*s* ✓
Harris set	2	4	2	7%	8%	19%	32%	4%	1378*s* ✓
Harris set (wait-free search)	2	4	2	8%	10%	17%	34%	3%	1066*s* ✓

Evaluation. We used `plankton` to automatically verify linearizability of fine-grained state-of-the-art set implementations from the literature: a lock-coupling set [Herlihy and Shavit 2008, Chapter 9.5], the Lazy set [Heller et al. 2005], FEMRS tree [Feldman et al. 2018] which is a variation of the contention-friendly binary tree [Crain et al. 2013, 2016], Vechev&Yahav 2CAS set [Vechev and Yahav 2008, Figures 8 and 9], Vechev&Yahav CAS set [Vechev and Yahav 2008, Figure 2], ORVYY set [O'Hearn et al. 2010], Michael set [Michael 2002], Harris set [Harris 2001], and a variation with wait-free `search` of the Michael and Harris set algorithms. For the FEMRS tree, `plankton` cannot handle the maintenance operations because they have updates with an unbounded footprint that is not traversed. This is a limitation of our current future reasoning. However, we are not aware of any other tool that can automatically verify even this simplified version of FEMRS trees. Also, `plankton` is the first tool to automate hindsight reasoning for the Harris set.

The results are summarized in Table 1. The first three columns of the table list (i) the number of iterations until the fixpoint $\mathbb{I}_{lfp}$ is reached, (ii) the size of $\mathbb{I}_{lfp}$, and (iii) the number of future candidates. The next five columns list the percentage of runtime spent on (iv) rule COM-SEM, (v) future reasoning, (vi) history reasoning, (vii) joins, and (viii) applying interferences. The last column gives (ix) the overall runtime, averaged across 10 runs, and the linearizability verdict (success is marked with ✓). Across all benchmarks we observe that two iterations are sufficient to reach the fixpoint $\mathbb{I}_{lfp}$: the first iteration discovers all interferences, the second iteration confirms that none are missing. This is remarkable because the first iteration uses $\mathbb{I}_0 \triangleq \emptyset$, i.e., considers the sequential setting. Further, we observe that most benchmarks spend significantly more time reasoning about the past than the future. The reason is twofold. (1) Introducing new futures either succeeds, meaning that a future candidate is resolved and can be ignored going forward, or it *fails fast*, which we attribute to Z3 finding counterexamples much faster than proving the validity of our SMT encoding of heap updates. (2) For histories, we do not have a heuristic identifying candidates. Instead, we eagerly introduce histories upon interference. We also apply hindsight reasoning eagerly. Lastly, we observe that the overall runtime tends to increase with the nesting depth and complexity of loops, as `plankton` requires several loop iterations (oftentimes between 3 and 5) to find an invariant. A proper investigation of how finding loop invariants affects the overall runtime is future work.

We also stress-tested `plankton` with faulty variations of the benchmarks. All buggy benchmarks failed verification. Note that `plankton` does not implement error explanation techniques, which are beyond the scope of the present paper.

9 RELATED WORK

Program logics with history and prophecy. Program logics have been extended by mechanisms for temporal reasoning in various ways [Abadi and Lamport 1991; Bell et al. 2010; Delbianco et al. 2017; Fu et al. 2010; Gotsman et al. 2013; Hemed et al. 2015; Liang and Feng 2013; Manna and Pnueli 1995; Parkinson et al. 2007; Schneider 1997; Sergey et al. 2015].

The work closest to ours is HLRG [Fu et al. 2010], a separation logic based on local rely-guarantee [Feng 2009] that tracks and reasons about history information, and its variation [Gotsman et al. 2013]. The separation algebra behind HLRG is constructed like ours. The focus of [Fu et al. 2010; Gotsman et al. 2013], however, are temporal operators in the assertion language and means of reasoning about them in the program logic. We only have now and past, but add the ability to propagate information between them. The simplicity of our approach enables automation ([Fu et al. 2010; Gotsman et al. 2013] is not even mechanized, as far as we know). A minor difference is that we work over general separation algebras to integrate flows [Krishna et al. 2018, 2020b] easily and make the requirement of frameability explicit.

A program logic with temporal information based on different principles appears in [Delbianco et al. 2017; Sergey et al. 2015]. There, histories are sub-computations represented by timestamped sets of events. The product of histories is disjoint union. While highly expressive, we are not aware of implementations of the approach. Since our goal is automated proof construction, we strive for assertions that are simple to prove, instead.

A separation logic for proving producer-consumer applications is proposed in [Bell et al. 2010]. The logic uses history but is domain-specific and provides no mechanism to reason about the temporal development. A non-blocking stack with memory reclamation is verified in [Parkinson et al. 2007]. The proof relies on history information stored in auxiliary variables and manipulated by ghost code. The ghost code is justified by informal arguments (outside the program logic). We do not consider memory reclamation as it can be verified separately [Meyer and Wolff 2019, 2020]. Beyond linearizability, history variables have recently been used to give specs to non-linearizable objects [Hemed et al. 2015].

Several separation logics have been extended with prophecy variables [Jung et al. 2020; Liang and Feng 2013], which complement history-based reasoning with a mechanism to speculate about future events. However, prophecies are not well-suited for automatic proofs because they rely on backward reasoning [Bouajjani et al. 2017].

History reasoning has been used early on in program verification [Abadi and Lamport 1991]. In program logics [Manna and Pnueli 1995; Schneider 1997], the focus has been on causality formulas which, in our notation, take the form $\_p \Rightarrow \odot q$. Our history reasoning is more flexible, in particular incorporates hindsight reasoning (see below), and inherits the benefits of modern separation logics.

Overall, existing program logics with history are heavier than ours while missing the important trick of communicating information from the current state to the past by means of logical variables shared between the two.

Hindsight reasoning. The idea of propagating information from the current state into the past is inspired by the recent hindsight theory [Feldman et al. 2018, 2020; Lev-Ari et al. 2015; O'Hearn et al. 2010]. Hindsight lemmas ensure that information about a data structure obtained by sequential reasoning (typically the reachability of keys) remains valid for concurrent executions. The argument behind such results is that the existence of a sequentially-reachable state implies the existence of a related concurrently-reachable state in the past. The implication requires that updates to the structure do not interfere with the reachability condition one tries to establish (*forepassing condition* in [Feldman et al. 2020]).

So far, hindsight reasoning has been limited to pencil-and-paper proofs, with the exception of the `poling` tool [Zhu et al. 2015]. `poling` automates the specific hindsight lemma of O'Hearn et al. [2010]. Unlike histories, it does not immediately generalize to other forms of retrospective reasoning, like [Feldman et al. 2018, 2020; Lev-Ari et al. 2015].

Our program logic makes past states explicit and can be understood as a formal framework in which to execute hindsight reasoning. The sequential-to-concurrent lifting of hindsight matches the thread-modular nature of our logic. This brings several benefits. For our engine, hindsight arguments provide a *strategy* for finding history information. For hindsight theory, it would be interesting to execute their arguments in our framework. This not only gives the classical benefits of program logics like precision and mechanization resp. automation. One also inherits the other features of our logic. We found futures indispensable to prove the Harris set.

Futures. Our futures are nothing but Hoare triples in separation logic (with separating implication), and their use as assertions is well-known from program logics like Iris [Jung et al. 2018]. What we add is the observation that futures capture complex heap updates by iteratively combining futures of small updates found during the traversal preparing the complex update. This iterative combination is the key novelty of our development. It allows us to reason about updates of unbounded heap regions by means of updates of finite regions.

Futures are inspired by atomic triples in TADA [da Rocha Pinto et al. 2014] and its predecessor CAP [Dinsdale-Young et al. 2010]. Atomic triples are specifications that justify an abstract notion of atomicity in a way that makes them suitable for constructing nested modules based on local specifications. Futures, although composed during the traversal, are also executed atomically upon the CAS. The difference is that an atomic triple is static in that the specification of the module is not meant to change during the execution. A future is instead built up during a loop that prepares the preceding update specified by the future.

On the more technical side, we note that futures carry resources whereas the nested Hoare triples in Iris are persistent [Jung et al. 2018]. The point is that our logic is not higher-order nor supports function pointers/recursion through the heap, therefore does not need step-indexed semantics [Appel and McAllester 2001] nor has to be careful about operators [Schwinghammer et al. 2009].

A method for automatically handling updates affecting unbounded heap regions is proposed in [Ter-Gabrielyan et al. 2019], however, their method is tailored towards reachability. Being Hoare triples, our futures are not restricted to a specific class of properties.

Automation. There is a considerable body of work on the automated verification of concurrent data structures. For static linearization points, there are tools [Abdulla et al. 2013] and well-chosen abstract domains [Abdulla et al. 2018]. For dynamic linearization points, there are reductions to safety verification [Bouajjani et al. 2013, 2015, 2017]. Common to these works is that, in the end, they rely on a state-space search whereas our approach reasons in a program logic. Notably, the `poling` tool [Zhu et al. 2015] extends `cave` [Vafeiadis 2009, 2010a,b] to support dynamic linearization points, e.g., to verify intricate stacks and queues (which we do not support because they are no search structures). Related is also [Itzhaky et al. 2014] in the sense that flows in particular can express heap paths. But we are not interested in verification condition generation and complete reductions to SMT, but rather proof generation, including invariant synthesis.

Other promising tools automating program logics include `Starling` [Windsor et al. 2017], `Caper` [Dinsdale-Young et al. 2017], `Voila` [Wolf et al. 2021], and `Diaframe` [Mulder et al. 2022]. However, these are closer to proof-outline checkers when compared to our tool. In particular, they do not perform loop invariant and interference inference or try to identify linearization points. Instead, they target more complex logics that are not designed for ease of automation.

REFERENCES

Martín Abadi and Leslie Lamport. 1991. The Existence of Refinement Mappings. *Theor. Comput. Sci.* 82, 2 (1991), 253–284.

Parosh Aziz Abdulla, Frédéric Haziza, Lukás Holík, Bengt Jonsson, and Ahmed Rezine. 2013. An Integrated Specification and Verification Technique for Highly Concurrent Data Structures. In *TACAS (Lecture Notes in Computer Science, Vol. 7795)*. Springer, 324–338.

Parosh Aziz Abdulla, Bengt Jonsson, and Cong Quy Trinh. 2018. Fragment Abstraction for Concurrent Shape Analysis. In *ESOP (Lecture Notes in Computer Science, Vol. 10801)*. Springer, 442–471.

Andrew W. Appel and David A. McAllester. 2001. An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. Program. Lang. Syst.* 23, 5 (2001), 657–683.

Hans Bekic. 1984. Note on a Test Example for ALGOL 60 Compilers by D. E. Knuth. In *Programming Languages and Their Definition (Lecture Notes in Computer Science, Vol. 177)*. Springer, 1.

Christian J. Bell, Andrew W. Appel, and David Walker. 2010. Concurrent Separation Logic for Pipelined Parallelization. In *SAS (Lecture Notes in Computer Science, Vol. 6337)*. Springer, 151–166.

Ahmed Bouajjani, Michael Emmi, Constantin Enea, and Jad Hamza. 2013. Verifying Concurrent Programs against Sequential Specifications. In *ESOP (Lecture Notes in Computer Science, Vol. 7792)*. Springer, 290–309.

Ahmed Bouajjani, Michael Emmi, Constantin Enea, and Jad Hamza. 2015. On Reducing Linearizability to State Reachability. In *ICALP (2) (Lecture Notes in Computer Science, Vol. 9135)*. Springer, 95–107.

Ahmed Bouajjani, Michael Emmi, Constantin Enea, and Suha Orhun Mutluergil. 2017. Proving Linearizability Using Forward Simulations. In *CAV (2) (Lecture Notes in Computer Science, Vol. 10427)*. Springer, 542–563.

Cristiano Calcagno, Peter W. O'Hearn, and Hongseok Yang. 2007. Local Action and Abstract Separation Logic. In *LICS*. IEEE Computer Society, 366–378.

Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL*. ACM, 238–252.

Patrick Cousot and Radhia Cousot. 1979. Systematic Design of Program Analysis Frameworks. In *POPL*. ACM Press, 269–282.

Tyler Crain, Vincent Gramoli, and Michel Raynal. 2013. A Contention-Friendly Binary Search Tree. In *Euro-Par (Lecture Notes in Computer Science, Vol. 8097)*. Springer, 229–240.

Tyler Crain, Vincent Gramoli, and Michel Raynal. 2016. A Fast Contention-Friendly Binary Search Tree. *Parallel Process. Lett.* 26, 3 (2016), 1650015:1–1650015:17.

Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. 2014. TaDA: A Logic for Time and Data Abstraction. In *ECOOP (Lecture Notes in Computer Science, Vol. 8586)*. Springer, 207–231.

Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS (Lecture Notes in Computer Science, Vol. 4963)*. Springer, 337–340.

Germán Andrés Delbianco, Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. 2017. Concurrent Data Structures Linked in Time. In *ECOOP (LIPIcs, Vol. 74)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 8:1–8:30.

Edsger W. Dijkstra. 1976. *A Discipline of Programming*. Prentice-Hall.

Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew J. Parkinson, and Hongseok Yang. 2013. Views: compositional reasoning for concurrent programs. In *POPL*. ACM, 287–300.

Thomas Dinsdale-Young, Pedro da Rocha Pinto, Kristoffer Just Andersen, and Lars Birkedal. 2017. Caper - Automatic Verification for Fine-Grained Concurrency. In *ESOP (Lecture Notes in Computer Science, Vol. 10201)*. Springer, 420–447.

Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. 2010. Concurrent Abstract Predicates. In *ECOOP (Lecture Notes in Computer Science, Vol. 6183)*. Springer, 504–528.

Tayfun Elmas, Shaz Qadeer, Ali Sezgin, Omer Subasi, and Serdar Tasiran. 2010. Simplifying Linearizability Proofs with Reduction and Abstraction. In *TACAS (Lecture Notes in Computer Science, Vol. 6015)*. Springer, 296–311.

Yotam M. Y. Feldman, Constantin Enea, Adam Morrison, Noam Rinetzky, and Sharon Shoham. 2018. Order out of Chaos: Proving Linearizability Using Local Views. In *DISC (LIPIcs, Vol. 121)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 23:1–23:21.

Yotam M. Y. Feldman, Artem Khyzha, Constantin Enea, Adam Morrison, Aleksandar Nanevski, Noam Rinetzky, and Sharon Shoham. 2020. Proving highly-concurrent traversals correct. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 128:1–128:29.

Xinyu Feng. 2009. Local rely-guarantee reasoning. In *POPL*. ACM, 315–327.

Keir Fraser. 2004. *Practical lock-freedom*. Ph. D. Dissertation. University of Cambridge, UK.

Ming Fu, Yong Li, Xinyu Feng, Zhong Shao, and Yu Zhang. 2010. Reasoning about Optimistic Concurrency Using a Program Logic for History. In *CONCUR (Lecture Notes in Computer Science, Vol. 6269)*. Springer, 388–402.

Alexey Gotsman, Noam Rinetzky, and Hongseok Yang. 2013. Verifying Concurrent Memory Reclamation Algorithms with Grace. In *ESOP (Lecture Notes in Computer Science, Vol. 7792)*. Springer, 249–269.

Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan (Newman) Wu, Jérémie Koenig, Vilhelm Sjöberg, Hao Chen, David Costanzo, and Tahina Ramananandro. 2018. Certified concurrent abstraction layers. In *PLDI*. ACM, 646–661.

Timothy L. Harris. 2001. A Pragmatic Implementation of Non-blocking Linked-Lists. In *DISC (Lecture Notes in Computer Science, Vol. 2180)*. Springer, 300–314.

Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N. Scherer III, and Nir Shavit. 2005. A Lazy Concurrent List-Based Set Algorithm. In *OPODIS (Lecture Notes in Computer Science, Vol. 3974)*. Springer, 3–16.

Nir Hemed, Noam Rinetzky, and Viktor Vafeiadis. 2015. Modular Verification of Concurrency-Aware Linearizability. In *DISC (Lecture Notes in Computer Science, Vol. 9363)*. Springer, 371–387.

Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Shaz Qadeer. 2003. Thread-Modular Abstraction Refinement. In *CAV (Lecture Notes in Computer Science, Vol. 2725)*. Springer, 262–274.

Maurice Herlihy and Nir Shavit. 2008. *The art of multiprocessor programming*. Morgan Kaufmann.

Maurice Herlihy and J. D. Tygar. 1987. How to Make Replicated Data Secure. In *CRYPTO (Lecture Notes in Computer Science, Vol. 293)*. Springer, 379–391.

Shachar Itzhaky, Anindya Banerjee, Neil Immerman, Ori Lahav, Aleksandar Nanevski, and Mooly Sagiv. 2014. Modular reasoning about heap paths via effectively propositional formulas. In *POPL*. ACM, 385–396.

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.* 28 (2018), e20.

Ralf Jung, Rodolphe Lepigre, Gaurav Parthasarathy, Marianna Rapoport, Amin Timany, Derek Dreyer, and Bart Jacobs. 2020. The future is ours: prophecy variables in separation logic. *Proc. ACM Program. Lang.* 4, POPL (2020), 45:1–45:32.

Bernhard Kragl, Shaz Qadeer, and Thomas A. Henzinger. 2020. Refinement for Structured Concurrent Programs. In *CAV (1) (Lecture Notes in Computer Science, Vol. 12224)*. Springer, 275–298.

Siddharth Krishna, Nisarg Patel, Dennis E. Shasha, and Thomas Wies. 2020a. Verifying concurrent search structure templates. In *PLDI*. ACM, 181–196.

Siddharth Krishna, Dennis E. Shasha, and Thomas Wies. 2018. Go with the flow: compositional abstractions for concurrent data structures. *Proc. ACM Program. Lang.* 2, POPL (2018), 37:1–37:31.

Siddharth Krishna, Alexander J. Summers, and Thomas Wies. 2020b. Local Reasoning for Global Graph Properties. In *ESOP (Lecture Notes in Computer Science, Vol. 12075)*. Springer, 308–335.

Kfir Lev-Ari, Gregory V. Chockler, and Idit Keidar. 2015. A Constructive Approach for Proving Data Structures' Linearizability. In *DISC (Lecture Notes in Computer Science, Vol. 9363)*. Springer, 356–370.

Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-tree for new hardware platforms. In *ICDE*. IEEE Computer Society, 302–313.

Hongjin Liang and Xinyu Feng. 2013. Modular verification of linearizability with non-fixed linearization points. In *PLDI*. ACM, 459–470.

Zohar Manna and Amir Pnueli. 1995. *Temporal verification of reactive systems - safety*. Springer.

Roland Meyer and Sebastian Wolff. 2019. Decoupling lock-free data structures from memory reclamation for static analysis. *Proc. ACM Program. Lang.* 3, POPL (2019), 58:1–58:31.

Roland Meyer and Sebastian Wolff. 2020. Pointer life cycle types for lock-free data structures with memory reclamation. *Proc. ACM Program. Lang.* 4, POPL (2020), 68:1–68:36.

Maged M. Michael. 2002. High performance dynamic lock-free hash tables and list-based sets. In *SPAA*. ACM, 73–82.

Ike Mulder, Robbert Krebbers, and Herman Geuvers. 2022. Diaframe: Automated Verification of Fine-Grained Concurrent Programs in Iris. In *PLDI*. ACM.

Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. 2001. Local Reasoning about Programs that Alter Data Structures. In *CSL (Lecture Notes in Computer Science, Vol. 2142)*. Springer, 1–19.

Peter W. O'Hearn, Noam Rinetzky, Martin T. Vechev, Eran Yahav, and Greta Yorsh. 2010. Verifying linearizability with hindsight. In *PODC*. ACM, 85–94.

Susan S. Owicki and David Gries. 1976. An Axiomatic Proof Technique for Parallel Programs I. *Acta Informatica* 6 (1976), 319–340.

Matthew J. Parkinson, Richard Bornat, and Peter W. O'Hearn. 2007. Modular verification of a non-blocking stack. In *POPL*. ACM, 297–302.

Xavier Rival and Laurent Mauborgne. 2007. The trace partitioning abstract domain. *ACM Trans. Program. Lang. Syst.* 29, 5 (2007), 26.

Fred B. Schneider. 1997. *On Concurrent Programming*. Springer.

Jan Schwinghammer, Lars Birkedal, Bernhard Reus, and Hongseok Yang. 2009. Nested Hoare Triples and Frame Rules for Higher-Order Store. In *CSL (Lecture Notes in Computer Science, Vol. 5771)*. Springer, 440–454.

Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. 2015. Specifying and Verifying Concurrent Algorithms with Histories and Subjectivity. In *ESOP (Lecture Notes in Computer Science, Vol. 9032)*. Springer, 333–358.

Dennis E. Shasha and Nathan Goodman. 1988. Concurrent Search Structure Algorithms. *ACM Trans. Database Syst.* 13, 1 (1988), 53–90.

Arshavir Ter-Gabrielyan, Alexander J. Summers, and Peter Müller. 2019. Modular verification of heap reachability properties in separation logic. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 121:1–121:28.

Viktor Vafeiadis. 2008. *Modular fine-grained concurrency verification.* Ph. D. Dissertation. University of Cambridge, UK.

Viktor Vafeiadis. 2009. Shape-Value Abstraction for Verifying Linearizability. In *VMCAI (Lecture Notes in Computer Science, Vol. 5403)*. Springer, 335–348.

Viktor Vafeiadis. 2010a. Automatically Proving Linearizability. In *CAV (Lecture Notes in Computer Science, Vol. 6174)*. Springer, 450–464.

Viktor Vafeiadis. 2010b. RGSep Action Inference. In *VMCAI (Lecture Notes in Computer Science, Vol. 5944)*. Springer, 345–361.

Viktor Vafeiadis and Matthew J. Parkinson. 2007. A Marriage of Rely/Guarantee and Separation Logic. In *CONCUR (Lecture Notes in Computer Science, Vol. 4703)*. Springer, 256–271.

Martin T. Vechev and Eran Yahav. 2008. Deriving linearizable fine-grained concurrent objects. In *PLDI*. ACM, 125–135.

Matt Windsor, Mike Dodds, Ben Simner, and Matthew J. Parkinson. 2017. Starling: Lightweight Concurrency Verification with Views. In *CAV (1) (Lecture Notes in Computer Science, Vol. 10426)*. Springer, 544–569.

Felix A. Wolf, Malte Schwerhoff, and Peter Müller. 2021. Concise Outlines for a Complex Logic: A Proof Outline Checker for TaDA. In *FM (Lecture Notes in Computer Science, Vol. 13047)*. Springer, 407–426.

Hongseok Yang, Oukseh Lee, Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, and Peter W. O'Hearn. 2008. Scalable Shape Analysis for Systems Code. In *CAV (Lecture Notes in Computer Science, Vol. 5123)*. Springer, 385–398.

He Zhu, Gustavo Petri, and Suresh Jagannathan. 2015. Poling: SMT Aided Linearizability Proofs. In *CAV (2) (Lecture Notes in Computer Science, Vol. 9207)*. Springer, 3–19.

A FULL HISTORY PROOF FOR HARRIS' SET

Figure 9 shows the full proof outline for the linearizability of `search` from the Harris' set.

```
TravInv(N, M, l, ln, lmark, t) ≜ TInv(N, M, l, ln, lmark, t) ∗ Fut(M, l, ln, t)
Hist(P, t) ≜ ⟐ (`Node(t) ∗ (P ⇒ `(k ∈ KS(t))) ) ∗ (`mark(t) ⇒ mark(t)) ∗ key(t) = `key(t)

{ ∃N M. TravInv(N, M, l, ln, lmark, t) ∗ Hist(k ≤ `key(ln), ln) }
procedure traverse(k: K, l: N, ln: N, lmark: Bool, t: N) {
  val tn, tmark = atomic {t.next, t.mark}
  { TravInv(N, M, l, ln, lmark, t) ∗ Hist(k ≤ `key(ln), ln) }
  skip
  { TravInv(N, M, l, ln, lmark, t) ∗ Hist(k ≤ `key(ln), ln) ∗ (key(t) < k ⇒ Hist((k ≤ `key(tn) ∧ ¬tmark), tn)) }
  if (tmark) {
    { TravInv(N, M, l, ln, lmark, tn) ∗ Hist(k ≤ `key(ln), ln) }
    return traverse(k, l, ln, lmark, tn)
  } else if (t.key < k) {
    { TravInv(N, M, t, tn, tmark, tn) ∗ Hist(k ≤ `key(tn), tn) }
    return traverse(k, t, tn, tmark, tn)
  } else {
    { TravInv(N, M, l, ln, lmark, t) ∗ t ≠ head ∗ t ≠ l ∗ key(l) < k ≤ key(t) ∗ Hist(k ≤ `key(ln), ln) }
    return (l, ln, lmark, t)
} }
{ (l, ln, lmark, r). ∃N M. TravInv(N, M, l, ln, lmark, r) ∗ r ≠ head ∗ key(l) < k ≤ key(r) ∗ Hist(k ≤ `key(ln), ln) }

{ ∃N. Inv(N) ∗ −∞ < k < ∞ }
procedure find(k: K) : N * N {
  val hn, hmark = atomic {head.next, head.mark}
  { TravInv(N, {head, hn}, head, hn, hmark, hn) ∗ hn = next(head) }
  skip
  { TravInv(N, {head, hn}, head, hn, hmark, hn) ∗ Hist(k ≤ `key(hn), hn) }
  val l, ln, lmark, r = traverse(k, head, hn, hmark, hn)
  { TravInv(N, M, l, ln, lmark, r) ∗ r ≠ head ∗ key(l) < k ≤ key(r) ∗ Hist(k ≤ `key(ln), ln) }
  val succ = ln == r || atomic { // CAS
    l.next == ln && l.mark == lmark ? {
      { next(l) = ln ∗ ¬mark(l) ∗ { l, ln, t } ⊆ M ⊆ N ∗ Fut(M, l, ln, r) ∗ key(l) < k ≤ key(r) ∗ Inv(N) }
      l.next := r; true
      { Inv(N) ∗ { l, r } ⊆ N ∗ (key(l), ∞] ⊆ flow(r) ∗ key(l) < k ≤ key(r) }
      skip
      { Inv(N) ∗ { l, r } ⊆ N ∗ Hist(true, r) }
      } : false
  }
  { Inv(N) ∗ { l, r } ⊆ N ∗ (succ ⇒ Hist(true, r)) }
  if (succ && !r.mark) {
    { Inv(N) ∗ { l, r } ⊆ N ∗ ¬`mark(r) ∗ Hist(true, r) }
    return (l, r)
  } else find(k)
}
{ (l, r). ∃N. Inv(N) ∗ { l, r } ⊆ N ∗ ¬`mark(r) ∗ Hist(true, r) }
{ (l, r). ∃N. Inv(N) ∗ { l, r } ⊆ N ∗ key(r) = `key(r) ∗ ¬`mark(r) ∗ ⟐ (`Node(r) ∗ `(k ∈ KS(r))) }
```

Fig. 9. Linearizability proof outline for Harris' set `find`.

B ADDITIONAL MATERIAL FOR SECTION 3

Example 19. We use a standard heap semantics of separation logic as a running example throughout this section. We consider heaps consisting of objects mapping fields to values. Let Sel be a set of *field selectors*. Let further Val be a set of values, $Addr \subseteq Val$ an infinite set of *addresses*, and Var a set of *program variables*. A *stack* $\iota \in \Sigma_\mathsf{S} \triangleq Var \to Val$ is an assignment from program variables to values. A *heap graph* is a tuple $h = (X, sval)$ consisting of a finite set of nodes $X \subseteq Addr$ and a total valuation of the selectors $sval : X \times Sel \to Val$. Note that selectors may evaluate to nodes outside the heap graph. Let HG be the set of all heap graphs.

The composition of heap graphs is the disjoint union, $h_1 *_\mathsf{H} h_2 \triangleq (h_1.X \uplus h_2.X, h_1.sval \uplus h_2.sval)$, and the empty heap is the only unit, $\mathsf{emp}_\mathsf{H} \triangleq \{ (\emptyset, \emptyset) \}$. We extend the composition to pairs of stack and heap by defining $(\iota_1, h_1) *_\mathsf{SH} (\iota_2, h_2) \triangleq (\iota_1, h_1 *_\mathsf{H} h_2)$ if $\iota_1 = \iota_2$. We then define the global states as $(\Sigma_\mathsf{G}, *_\mathsf{G}, \mathsf{emp}_\mathsf{G}) \triangleq (\Sigma_\mathsf{H}, *_\mathsf{H}, \mathsf{emp}_\mathsf{H})$ and the local states as $(\Sigma_\mathsf{L}, *_\mathsf{L}, \mathsf{emp}_\mathsf{L}) \triangleq (\Sigma_\mathsf{S} \times \Sigma_\mathsf{H}, *_\mathsf{SH}, \Sigma_\mathsf{S} \times \mathsf{emp}_\mathsf{H})$. A state is a pair of global and local state whose heaps are disjoint, $\Sigma \triangleq \{ (h_\mathsf{G}, (\iota_\mathsf{L}, h_\mathsf{L})) \mid h_\mathsf{G} \#_\mathsf{H} h_\mathsf{L} \}$.

Example 20. Continuing with Example 19, we define the semantics of a field update command $l.\mathtt{next} := r$ that assigns the field `next` of the object at address l to value r as follows:

$$[\![l.\mathtt{next} := r]\!](h_\mathsf{G}, (\iota, h_\mathsf{L})) \triangleq \begin{cases} \{ (h_\mathsf{G}, (\iota, h_\mathsf{L}[(\iota(l), \mathtt{next}) \mapsto \iota(r)])) \} & \text{if } \iota(l) \in h_\mathsf{L}.X \\ \{ (h_\mathsf{G}[(\iota(l), \mathtt{next}) \mapsto \iota(r)], (\iota, h_\mathsf{L})) \} & \text{if } \iota(l) \in h_\mathsf{G}.X \\ \mathsf{abort} & \text{otherwise .} \end{cases}$$

We lift $[\![l.\mathtt{next} := r]\!]$ to predicates in the expected way and also define $[\![l.\mathtt{next} := r]\!](\mathsf{abort}) = \mathsf{abort}$.

Proof of Lemma 1. Assume that $(\Sigma_\mathsf{G}, *_\mathsf{G}, \mathsf{emp}_\mathsf{G})$ and $(\Sigma_\mathsf{L}, *_\mathsf{L}, \mathsf{emp}_\mathsf{L})$ are separation algebras and assume that Σ is such that $\mathsf{emp}_\mathsf{G} \times \mathsf{emp}_\mathsf{L} \subseteq \Sigma \subseteq \Sigma_\mathsf{G} \times \Sigma_\mathsf{L}$, and Σ is closed under decomposition.

The laws of commutativity and units follow directly from the fact that $*$ is the component-wise lifting of $*_\mathsf{G}$ and $*_\mathsf{L}$ on Σ_G and Σ_L, and that $\mathsf{emp}_\mathsf{G} \times \mathsf{emp}_\mathsf{L} \subseteq \Sigma$.

To show associativity of $*$, let $\mathsf{s}_1, \mathsf{s}_2, \mathsf{s}_3 \in \Sigma$ such that $\mathsf{s}_2 \# \mathsf{s}_3$ and $\mathsf{s}_1 \# (\mathsf{s}_2 * \mathsf{s}_3)$. We have:

$$\begin{aligned} \mathsf{s}_1 * (\mathsf{s}_2 * \mathsf{s}_3) &= (\mathsf{s}_1.\mathsf{g} *_\mathsf{G} (\mathsf{s}_2.\mathsf{g} *_\mathsf{G} \mathsf{s}_3.\mathsf{g}), \mathsf{s}_1.\mathsf{l} *_\mathsf{L} (\mathsf{s}_2.\mathsf{l} *_\mathsf{L} \mathsf{s}_3.\mathsf{l})) \\ &= ((\mathsf{s}_1.\mathsf{g} *_\mathsf{G} \mathsf{s}_2.\mathsf{g}) *_\mathsf{G} \mathsf{s}_3.\mathsf{g}, (\mathsf{s}_1.\mathsf{l} *_\mathsf{L} \mathsf{s}_2.\mathsf{l}) *_\mathsf{L} \mathsf{s}_3.\mathsf{l}) = (\mathsf{s}_1 * \mathsf{s}_2) * \mathsf{s}_3 \end{aligned}$$

where the second equality follows by associativity of $*_\mathsf{G}$ and $*_\mathsf{L}$, and the third equality holds because Σ is closed under decomposition.

The other direction follows by commutativity. □

C FLOWS

The flow framework [Krishna et al. 2018, 2020b] enables local reasoning about inductive properties of graphs. Intuitively, the framework views a heap as a (data) flow graph that induces *flow constraints* prescribing how *flow values* are propagated along the pointer edges of the heap. A *flow* is a solution to these flow constraints and assigns a flow value to every node in the graph. For instance, one can define a *path-counting flow* that computes for every node x the number of paths that reach x from some dedicated root node. By imposing an appropriate invariant on the flow at every node, one can then express inductive properties about the heap. For example, using the path-counting flow one can capture structural invariants such as that a heap region forms a list, a tree, or a DAG.

Flow graphs are endowed with a separation algebra that extends disjoint composition of heap graphs $h_1 * h_2$ with a stronger definedness condition. This stronger condition ensures that the flow of $h_1 * h_2$ is also the union of the flows of h_1 and h_2. This separation algebra admits frame rules that capture whether a local update of the heap graph h_1 preserves the flow of the context h_2 (and hence the flow-based inductive invariant of $h_1 * h_2$).

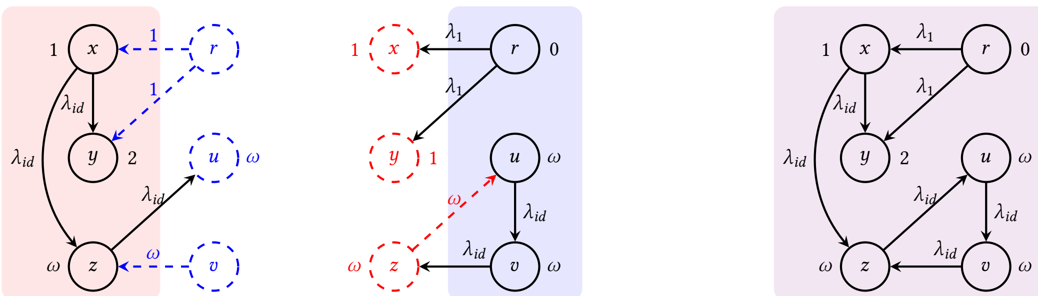

Fig. 10. Two flow constraints c_1 with $c_1.X = \{ x, y, z \}$ (left) and c_2 with $c_2.X = \{ r, u, v \}$ (center) for the path-counting flow monoid $\mathbb{N} \cup \{ \omega \}$. The edge label λ_{id} stands for the identity function and λ_1 for the constant 1 function. Omitted edges are labeled by the constant 0 function. Dashed edges represent the inflows. Nodes are labeled by their flow, respectively, outflow. The right side shows the composition $c_1 * c_2$.

In the following, we develop a new meta theory of flows that, unlike [Krishna et al. 2018, 2020b], is geared towards automatic inference of flow invariants, yielding a flow-based abstract domain for program analysis.

Separation Algebra. For a commutative monoid $(\mathbb{M}, +, 0)$ we define the binary relation $\leq$ on $\mathbb{M}$ as $n \leq m$ if there is $o \in \mathbb{M}$ with $m = n + o$. Flow values are drawn from a *flow monoid*, which is a commutative monoid for which the relation $\leq$ is a complete partial order. In the following, we fix a flow monoid $(\mathbb{M}, +, 0)$.

Let $\mathrm{Mon}(\mathbb{M} \to \mathbb{M})$ be the monotonic functions in $\mathbb{M} \to \mathbb{M}$. We lift $+$ and $\leq$ to functions $\mathbb{M} \to \mathbb{M}$ in the expected way. An iterated sum over an empty index set $\sum_{i \in \varnothing} m_i$ is defined to be 0.

A *flow constraint* is a tuple $c = (X, E, in)$ consisting of a finite set of variables $X \subseteq \mathit{Addr}$, a set of edges $E : X \times \mathit{Addr} \to \mathrm{Mon}(\mathbb{M} \to \mathbb{M})$ labeled by monotonic functions, and an *inflow* $in : (\mathit{Addr} \setminus X) \times X \to \mathbb{M}$. We use FC for the set of all flow constraints and denote the empty flow constraint by $c_\varnothing \triangleq (\varnothing, \varnothing, \varnothing)$.

We define two derived functions for flow constraints. The *flow* is the least function $\mathit{flow} : X \to \mathbb{M}$ satisfying

$$\mathit{flow}(x) = in(x) + \mathit{rhs}_x(\mathit{flow}) \quad \text{for all } x \in X\,.$$

Here, $in(x) \triangleq \sum_{y \in (\mathit{Addr} \setminus X)} in(y, x)$ is a monoid value and $\mathit{rhs}_x \triangleq \sum_{y \in X} E_{(y,x)}$ is a monotone function from $\mathrm{Mon}((X \to \mathbb{M}) \to \mathbb{M})$. We also define the *outflow* $\mathit{out} : X \times (\mathit{Addr} \setminus X) \to \mathbb{M}$ by $\mathit{out}(x, y) \triangleq E_{(x,y)}(\mathit{flow}(x))$.

Intuitively, a flow constraint c abstracts a heap graph h with domain $c.X$ whose contents induce the edge functions $c.E$. The inflow $c.in$ captures the contribution of h's context to $c.\mathit{flow}$ and the outflow $c.\mathit{out}$ is the contribution of c to the context's flow. In fact, if we abstract from the specific inflow of c, we can view c as a transformer from inflows to outflows. We make this intuition formally precise later. However, let us first discuss some examples.

Example 21. Path-counting flows are defined over the flow monoid of addition on natural numbers extended with the first limit ordinal, $\mathbb{N} \cup \{ \omega \}$. Here, $+$ is defined to be absorbing on ω. Figure 10 shows two flow constraints c_1 and c_2 for this flow monoid together with their flows and outflows. For example, we have $c_1.\mathit{flow}(y) = c_1.in(r, y) + \lambda_{id}(c_1.\mathit{flow}(x)) = 1 + 1 = 2$. The flow of each node corresponds to the number of paths starting from r that reach the node in the combined graph (shown on the right).

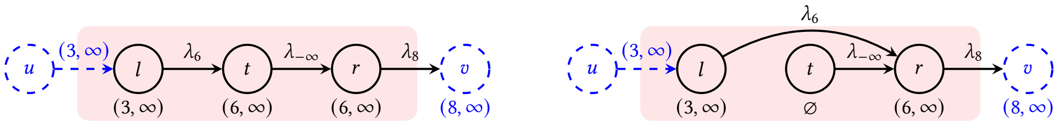

Fig. 11. Two flow constraints c_1 (left) and c_2 (right) with $c_1.X = c_2.X = \{ l, t, r \}$ for the keyset flow monoid $\mathbb{P}(\mathbb{Z} \cup \{ -\infty, \infty \})$. The edge label λ_k for a key k denotes the function $\lambda m.\, (m \setminus [-\infty, k])$.

Example 22. For our linearizability proofs of concurrent search structures we use a flow that labels every data structure node x with its *inset*, the set of keys k' such that a thread searching for k' traverses the node x. That is, the flow monoid is sets of keys, $\mathbb{P}(\mathbb{Z} \cup \{ -\infty, \infty \})$, with set union as addition. Figure 11 shows two flow constraints that abstract potential states of the Harris list. The idea is that an edge leaving a node x that stores a key k is labeled by the function λ_k. This is because a search for $k' \in \mathbb{Z}$ will traverse the edge leaving x iff $k < k'$ or x is marked. In the figure, l and r are assumed to be unmarked, storing keys 6 and 8, respectively. Node t is assumed to be marked. Flow constraint c_2 can be thought as being obtained from c_1 by physically unlinking the marked node t.

To define the composition of flow constraints, $c_1 * c_2$, we proceed in two steps. We first define an auxiliary composition that may suffer from *fake* flows, local flows that disappear in the composition. The actual composition then restricts the auxiliary composition to rule out such fake flows. Definedness of the auxiliary composition requires disjointness of the variables in c_1 and c_2. Moreover, the outflow of one constraint has to match the inflow expectations of the other:

$$\begin{aligned} c_1 \,\#\#\, c_2 \;\text{ if }\; & c_1.X \cap c_2.X = \varnothing \\ & \text{and } \forall x \in c_1.X.\ \forall y \in c_2.X c_1.out(x, y) = c_2.in(x, y) \wedge c_2.out(y, x) = c_1.in(y, x)\,. \end{aligned}$$

The auxiliary composition removes the inflow that is now provided by the other component, denoted by $c_1.in'$ and $c_2.in'$:

$$\begin{aligned} c_1 \uplus c_2 &\triangleq (c_1.X \uplus c_2.X, c_1.E \uplus c_2.E, c_1.in' \uplus c_2.in') \\ c_i.in'(x, y) &\triangleq (x \in c_{3-i}.X \wedge c_{3-i}.out(x, y) \neq 0) \;?\; 0 : c_i.in(x, y). \end{aligned}$$

To rule out fake flows, we incorporate a suitable equality on the flows into the definedness for the composition:

$$c_1 \,\#\, c_2 \quad \text{if} \quad c_1 \,\#\#\, c_2 \quad \wedge \quad c_1.flow \uplus c_2.flow = (c_1 \uplus c_2).flow.$$

Only if the latter equality holds, do we have the composition $c_1 * c_2 \triangleq c_1 \uplus c_2$. It is worth noting that $c_1.flow \uplus c_2.flow \geq (c_1 \uplus c_2).flow$ always holds. What definedness really asks for is the reverse inequality.

Example 23. The right side of Figure 10 shows the composition $c_1 * c_2$ of the flow constraints c_1 (left) and c_2 (center). Note that if the edge (x, z) is removed from c_1, then the composition is no longer defined since the flow of ω in the cycle formed by z, u, and v becomes a fake flow that vanishes in the flow constraint $c_1 \uplus c_2$ (the flow for z, u, and v becomes 0).

Lemma 24. *$(FC, *, \{ c_\varnothing \})$ is a separation algebra.*

Flow Graphs. Our heap model is a product between heap graphs and flow constraints. The flow component is meant to provide ghost information about the heap graph component. As we are interested in automating the reasoning, we will not modify the ghost information by ghost code but provide a mechanism to generate the flow constraint from the heap graph. This allows us to keep the flow constraint largely implicit.

Heap graphs $h \in HG$ are as defined in Example 19.

Lemma 25. *$(HG, *, \{ h_\emptyset \})$ is a separation algebra.*

To relate heap graphs and flow constraints, we classify selectors as data and pointer selectors, $Sel = DSel \uplus PSel$, and also write a heap graph $(X, sval)$ as $(X, dval, pval)$. Only pointer selectors will induce edges in a flow constraint via which we forward flow values. The data selectors will be used to determine the edge function. To this end, we assume to be given a generator for the edge functions:

$$gen : PSel \to (DSel \to Val) \to \mathrm{Mon}(\mathbb{M} \to \mathbb{M}) \ .$$

It takes a pointer selector and a valuation of the data selectors and returns a monotonic function over the monoid of flow values. The generator is a parameter to our development, like the flow monoid and the sets of data and pointer selectors.

Example 26. For the Harris list we have a single pointer selector `next` and data selectors `mark` and `key` storing the mark bit and key of a node, respectively.[1] In order to obtain the edge functions that abstract the Harris list as described in Example 22, we use the following generator:

$$gen(\mathtt{next}, dval) \triangleq dval(\mathtt{key}) = -\infty \ ? \ \lambda m.\, (-\infty, \infty] \ : \ \lambda_{dval(\mathtt{key})}$$

Recall that the condition $dval(\mathtt{key}) = -\infty$ only holds for the `head` node. We use it here to fix the flow leaving `head` to the constant interval $(-\infty, \infty]$.

A *flow graph* is a pair $fg = (h, c)$ consisting of a heap graph h and a flow constraint c so that the nodes of the heap graph and flow constraint coincide, $h.X = c.X$, and so that $c.E$ is induced by $h.pval$ and $h.dval$ via gen. Being induced means that for all $x \in c.X$ and $y \in Addr$:

$$c.E(x, y) \quad = \sum_{\substack{\mathsf{psel} \in PSel \\ y = h.pval(x, \mathsf{psel})}} gen(\mathsf{psel}, h.dval(x)).$$

Recall that empty sums are defined to be 0. The edge function iterates over all pointer selectors that lead from the source to the target node. It sums up the monoid functions given by the generator. Note that the edge functions are independent of the target node.

Example 27. Assuming the generator gen from the previous example, (h_1, c_1) is a flow graph where c_1 is as depicted in Figure 11:

$$\begin{aligned} h_1.pval &= \{ (l, \mathtt{next}) \mapsto t, (t, \mathtt{next}) \mapsto r, (r, \mathtt{next}) \mapsto v \} \\ h_1.dval &= \{ (l, mark) \mapsto false, (t, mark) \mapsto true \} \\ &\quad \uplus \{ (r, mark) \mapsto false \} \\ &\quad \uplus \{ (l, key) \mapsto 6, (t, key) \mapsto 7, (r, key) \mapsto 8 \} \ . \end{aligned}$$

We use FG for the set of all flow graphs. It is a subset of the product separation algebra between the heap graphs and the flow constraints, $FG \subseteq HG \times FC$. This gives flow graphs a notion of definedness: $fg_1 \# fg_2$, if $fg_1.h \# fg_2.h$ and $fg_1.c \# fg_2.c$. They also inherit the component-wise composition in case definedness holds: $fg_1 * fg_2 \triangleq (fg_1.h * fg_2.h, fg_1.c * fg_2.c)$. With this definition, we obtain a submonoid.

Lemma 28. *$(FG, *, \{ (h_\emptyset, c_\emptyset) \})$ is a separation algebra.*

[1] Technically, the mark bit is encoded in the value of `next`. We here treat them separately for the sake of exposition.

Commands and Frame-Preserving Updates. Let (h_1, c_1) be a flow graph and suppose that a command com updates the heap graph h_1 to a new heap graph h_2. Then the heap update induces a corresponding update of the flow constraint c_1 to the flow constraint c_2 generated from h_2. For instance, consider the flow graph (h_1, c_1) from Example 27. The heap h_2 obtained by executing the command $l.\mathtt{next} := r$ generates the flow constraint c_2 shown in Figure 11.

In order to ensure that the semantics of commands satisfies the locality condition (LocCom) assumed in §3, which is critical for the soundness of the FRAME rule, we need to ensure that the induced updates of the flow constraints are *frame-preserving* with respect to flow constraint composition. That is, if we have $c_1 \# c$ and we exchange c_1 by c_2, does $c_2 \# c$ still hold? Intuitively, $c_2 \# c$ still holds if c_1 and c_2 transform inflows to outflows in the same way.

Formally, for a flow constraint c we define its *transfer function* $tf(c)$ mapping inflows to outflows,

$$tf(c) : ((Addr \setminus X) \times X \to \mathbb{M}) \to X \times (Addr \setminus X) \to \mathbb{M}\,,$$

by $tf(c)(in') \triangleq c[in \mapsto in'].out$. For a given inflow in, we may also write $tf(c_1) =_{in} tf(c_2)$ to mean that for all inflows $in' \leq in$, we have $tf(c_1)(in') = tf(c_2)(in')$.

For example, we have $tf(c_1) =_{c_1.in} tf(c_2)$ for the flow constraints c_1 and c_2 in Figure 11. This is due to $c_1.in(x) = \emptyset$ for all $x \neq l$ and $\lambda_6 \circ \lambda_{-\infty} = \lambda_6$. Observe that c_2 composes with any frame c that c_1 composes with. This captures the fact that physically unlinking the marked node t does not affect the remaining state of the data structure. The following lemma generalizes this observation. Its proof relies on Bekic's lemma [Bekic 1984] and can be found in Appendix H.

Lemma 29. *Assume $c_1 \# c$, $c_1.X = c_2.X$, $c_1.in = c_2.in$, and $tf(c_1) =_{c_1.in} tf(c_2)$. Then (i) $c_2 \# c$ and (ii) $tf(c_1 * c) =_{(c_1 * c).in} tf(c_2 * c)$.*

In Appendix D we develop a semantics of a simple imperative programming language that satisfies the locality condition assumed in §3. Global states Σ_G and local states Σ_L are obtained by taking the product between flow graphs and variable valuations. The variable valuations track the values of both (immutable) program variables as well as logical variables. The construction of the state monoid $\Sigma \subseteq \Sigma_\mathsf{G} \times \Sigma_\mathsf{L}$ is mostly standard and similar to the one discussed in Example 19. The semantics aborts a heap update com if its induced flow constraint update does not satisfy the frame preservation condition identified in Lemma 29.

D INSTANTIATION

In this section, we instantiate the developed program logic to a simple imperative programming language whose states consist of flow graphs.

In the following, we write flow graphs as $fg = (X, sval, in)$. This goes without loss of generality since the nodes coincide for the heap graph and the flow constraint, and the edge function of the flow constraint is induced by the data and pointer selector valuations. For notational convenience, we also access the flow value by $fg.flow$ rather than $fg.c.flow$.

D.1 States

Valuations of program variables Y, or simply *stacks*, stem from $Val(Y) \triangleq Y \to \mathbb{N}$. Stacks are never split, $val_1 \# val_2$ is defined as $val_1 = val_2$.

Lemma 30. *$(Val(Y), *, Val(Y))$ is a separation algebra.*

As we divide states into a global and a local component, we distinguish between global program variables from the finite set *GPVar* and local variables from the disjoint and also finite set *LPVar*. We use $px, qx \in PVar = GPVar \uplus LPVar$ to refer to variables that are either global or local. The separation algebra of global states is the product $\Sigma_\mathsf{G} \triangleq FG \times Val(GPVar)$. Similarly, the separation algebra of local states is the product $\Sigma_\mathsf{L} \triangleq FG \times Val(LPVar)$. The set of states is a subset of the product separation algebra:

$$\Sigma \;\triangleq\; \{\, (\mathsf{g}, \mathsf{l}) \in \Sigma_\mathsf{G} \times \Sigma_\mathsf{L} \;\mid\; \mathit{wellsplit}(\mathsf{g}, \mathsf{l}) \,\}.$$

Here, the predicate $\mathit{wellsplit}(\mathsf{g}, \mathsf{l})$ says that it is possible to combine the global and the local heap. Moreover, the global state does not point into the local heap. Formally,

$$\begin{aligned}\mathit{wellsplit}(\mathsf{g}, \mathsf{l}) \quad \text{if} \quad & \mathsf{g}.\mathit{fg} \mathbin{\#} \mathsf{l}.\mathit{fg} \;\;\wedge\;\; \forall px \in GPVar.\;\; \mathsf{g}.\mathit{val}(px) \notin \mathsf{l}.\mathit{fg}.X \;\;\wedge \\ & \forall a \in \mathsf{g}.\mathit{fg}.X.\;\; \forall \mathsf{sel} \in Sel.\;\; \mathsf{g}.\mathit{fg}.\mathit{sval}(a, \mathsf{sel}) \notin \mathsf{l}.\mathit{fg}.X\;.\end{aligned}$$

The assumptions made by Lemma 1 follow from the next lemma.

Lemma 31. *For all* $(\mathsf{g}, \mathsf{l}) \in \mathsf{emp}_\mathsf{G} \times \mathsf{emp}_\mathsf{L}$, $\mathit{wellsplit}(\mathsf{g}, \mathsf{l})$ *holds. Furthermore, for all* $\mathsf{g}_1, \mathsf{g}_2 \in \Sigma_\mathsf{G}$ *and* $\mathsf{l}_1, \mathsf{l}_2 \in \Sigma_\mathsf{L}$*, if we have* $\mathit{wellsplit}(\mathsf{g}_1 *_\mathsf{G} \mathsf{g}_2, \mathsf{l}_1 *_\mathsf{L} \mathsf{l}_2)$*, then also* $\mathit{wellsplit}(\mathsf{g}_1, \mathsf{l}_1)$*.*

Corollary 32. $(\Sigma, *, \mathsf{emp}_\mathsf{G} \times \mathsf{emp}_\mathsf{L})$ *is a separation algebra.*

To fix the notation, we write $\mathsf{s}.\mathit{fg}$ for $\mathsf{s.g}.\mathit{fg} * \mathsf{s.l}.\mathit{fg}$ and similarly $\mathsf{s}.\mathit{val}$ for $\mathsf{s.g}.\mathit{val} \uplus \mathsf{s.l}.\mathit{val}$. Moreover, we denote $\mathsf{s}.\mathit{fg}.h$ by $\mathsf{s}.h$ and introduce a similar short-hand for $\mathsf{s}.\mathit{fg}.c$.

When we modify the valuations of program variables and selectors, the result is not necessarily well-split. We introduce the following function that turns a pair of global and local state into a state that is well-split:

$$\mathit{split} : \Sigma_\mathsf{G} \times \Sigma_\mathsf{L} \rightharpoonup \Sigma.$$

The idea is to move the nodes from the local heap that are reachable from the shared state to the shared heap. The function is partial because the nodes from the local heap may in turn point outside the state, in which case the effect would not be limited to the state. Definedness of the function is as follows:

$$\begin{aligned}(\mathsf{g}, \mathsf{l}) \mathbin{\#} \mathit{split} \quad \text{if} \quad \mathsf{g}.\mathit{fg} \mathbin{\#} \mathsf{l}.\mathit{fg} \quad \text{and} \quad & \forall a \in \mathsf{l}.\mathit{fg}.X\; \forall px \in GPVar\;\; \forall b \in \mathsf{g}.\mathit{fg}.X\;\; \forall \mathsf{sel} \in Sel. \\ & \qquad px \text{ reaches } a \;\;\vee\;\; b \text{ reaches } a \\ & \qquad \Rightarrow \mathsf{l}.\mathit{fg}.\mathit{sval}(a, \mathsf{sel}) \in \mathsf{l}.\mathit{fg}.X \uplus \mathsf{g}.\mathit{fg}.X\end{aligned}$$

The reachability predicate *px reaches a* resp. *b reaches a* is defined by either directly referencing address a (in the case of b through a selector) or by referencing an address that in turn reaches a. The formal definition is via a least fixed point. If defined, we set $\mathit{split}(\mathsf{g}, \mathsf{l}) \triangleq \mathsf{s}$ with $\mathsf{s}.\mathit{fg} = \mathsf{g}.\mathit{fg} * \mathsf{l}.\mathit{fg}$, $\mathsf{s}.\mathit{val} = \mathsf{g}.\mathit{val} \uplus \mathsf{l}.\mathit{val}$, $\mathsf{g}.\mathit{fg}.X \subseteq \mathsf{s.g}.\mathit{fg}.X$. Moreover, $\mathsf{s.l}.\mathit{fg}$ is the maximal (wrt. inclusion on $\mathsf{s}.\mathit{fg}.X$) flowgraph that satisfies these constraints. The function is well-defined in that it uniquely determines a state and the state does exist. To see this, note that the state can be obtained by a least fixed point iteration that moves local nodes into the shared heap.

Lemma 33. $\mathit{split}(\mathsf{g}, \mathsf{l})$ *is well-defined.*

We will need that *split* is a local function. To formulate this, we define state modifications of the form $[px \mapsto a]$ and $[(a, \mathsf{sel}) \mapsto b]$. This will modify the corresponding component in the state. The component will be clear from the type of the modification. For example, $\mathsf{s}[(a, \mathsf{sel}) \mapsto b]$ will modify the valuation *sval* at the selector sel of address a. Since the flow graphs in the global and in the local state can be composed, only one of them will hold the address.

The modification itself should also have a local effect. To formulate this, we introduce updates

$$\mathit{up} : \mathbb{N} \times \mathbb{N} \rightharpoonup \mathrm{Mon}(\mathbb{M} \to \mathbb{M})$$

as commands for changing flow constraints c, which we then subsequently relate to the updates of heap graphs. The application of the update yields the new flow constraint

$$c[up] \triangleq (c.X, E, c.in)$$

where the edges are defined by $E(x,y) \triangleq up(x,y)$ if up is defined on (x,y) and $E(x,y) \triangleq c.E(x,y)$ otherwise. Note that neither the nodes nor the inflow change. In Figure 11, e.g., we have $c_2 = c_1[up]$ for $up = \{ (l,t) \mapsto \lambda_\infty, (l,r) \mapsto \lambda_6 \}$.

When considering updates as commands, we let the semantics abort should the update not be defined on the flow constraint or should it change the transformer. Using Lemma 29, this makes the semantics suitable for framing:

$$[\![up]\!](c) \triangleq \begin{cases} c[up] & \text{if } \operatorname{dom}(up) \subseteq c.X \times \mathbb{N} \wedge \mathit{tf}(c) =_{c.in} \mathit{tf}(c[up]) \\ \text{abort} & \text{otherwise.} \end{cases}$$

The interplay between the semantics of updates and composition is captured by the following lemma. It ensures that the locality condition assumed in § 3 is satisfied, once we tie flow constraint updates to updates on the heap graphs from which the flow constraints are derived.

Lemma 34. *Let $[\![up]\!](c) \neq$ abort and $c \# c'$. Then we have:*
(i) $[\![up]\!](c) \# c'$ *and*
(ii) $[\![up]\!](c) * c' = (c * c')[up]$ *and*
(iii) $[\![up]\!](c * c') \neq$ *abort and*
(iv) $[\![up]\!](c * c') = [\![up]\!](c) * c'$.

We can now associate with a modification $[(a, \mathrm{sel}) \mapsto b]$ an update up of the state's flow constraint $\mathrm{s}.c$. Here, we use $\mathrm{s} \# [(a, \mathrm{sel}) \mapsto b]$ to denote the fact that the update induced by the modification does not abort on the flow constraint, $[\![up]\!](\mathrm{s}.c) \neq$ abort.

Lemma 35. *If we have*
- $\mathrm{s} \# \mathrm{s}'$,
- $a \in \mathrm{s.g}.\mathit{fg}.X \Rightarrow b \in \mathrm{s}.\mathit{fg}.X$,
- $\mathrm{s} \# [(a, \mathrm{sel}) \mapsto b]$, *and*
- $\mathrm{s}[(a, \mathrm{sel}) \mapsto b] \# \mathit{split}$,

then we get
- $(\mathrm{s} * \mathrm{s}') \# [(a, \mathrm{sel}) \mapsto b]$,
- $(\mathrm{s} * \mathrm{s}')[(a, \mathrm{psel}) \mapsto b] \# \mathit{split}$, *and*
- $\mathit{split}((\mathrm{s} * \mathrm{s}')[(a, \mathrm{sel}) \mapsto b]) = \mathit{split}(\mathrm{s}[(a, \mathrm{sel} \mapsto b]) * \mathrm{s}'$.

The latter separating conjunction is defined.

Proof. We have $(\mathrm{s} * \mathrm{s}') \# [(a, \mathrm{sel}) \mapsto b]$ by Lemma 34(iii).
Moreover, $(\mathrm{s} * \mathrm{s}')[(a, \mathrm{sel}) \mapsto b] = \mathrm{s}[(a, \mathrm{sel}) \mapsto b] \times \mathrm{s}'$ by Lemma 34(iv). The product on the right hand side of the previous equality denotes the separating conjunction applied componentwise, on the global and on the local state. It does not take into account well-splitting. If $a \in \mathrm{s.l}.\mathit{fg}.X$, there is no splitting to be done and we obtain

$$\begin{aligned} & \mathit{split}((\mathrm{s} * \mathrm{s}')[(a, \mathrm{sel}) \mapsto b]) \\ (\text{ No splitting }) \quad &= (\mathrm{s} * \mathrm{s}')[(a, \mathrm{sel}) \mapsto b] \\ (\text{ Above }) \quad &= \mathrm{s}[(a, \mathrm{sel}) \mapsto b] \times \mathrm{s}' \\ (\ \mathrm{s} \# \mathrm{s}', a \in \mathrm{s.l}.\mathit{fg}\ ) \quad &= \mathrm{s}[(a, \mathrm{sel}) \mapsto b] * \mathrm{s}' \\ (\text{ No splitting }) \quad &= \mathit{split}(\mathrm{s}[(a, \mathrm{sel}) \mapsto b]) * \mathrm{s}' \ . \end{aligned}$$

If $a \in \mathsf{s.g.}\mathit{fg}.X$, we have $b \in \mathsf{s.}\mathit{fg}.X$. Since s, s' are states, they are well-split and so the global part of each state does not point into the local heap of that state. Moreover, since $\mathsf{s} \# \mathsf{s}'$ by the assumption, the global part of state s' does not point into the local heap of s. With the same argument, the global part of s does not point into the local heap of s'.
If $b \in \mathsf{s.g.}\mathit{fg}.X$, this continues to hold for $\mathsf{s}[(a, \mathsf{sel}) \mapsto b]$.
If $b \in \mathsf{s.l.}\mathit{fg}.X$, this continues to hold for $\mathsf{s}[(a, \mathsf{sel}) \mapsto b]$ as well, because $\mathsf{s}[(a, \mathsf{sel}) \mapsto b] \# \mathit{split}$. The definedness says that no local node from $\mathsf{s}[(a, \mathsf{sel}) \mapsto b]$ reachable from the global part points outside $\mathsf{s}[(a, \mathsf{sel}) \mapsto b]$, in particular b. Note that traversing b is the only way to reach a local node from the global part in $\mathsf{s}[(a, \mathsf{sel}) \mapsto b]$. With the above discussion, splitting the state

$$(\mathsf{s} * \mathsf{s}')[(a, \mathsf{sel}) \mapsto b] = \mathsf{s}[(a, \mathsf{sel}) \mapsto b] \times \mathsf{s}'$$

will only move reachable nodes from $\mathsf{s.l.}\mathit{fg}$ through b to the global part. Since $\mathsf{s}[(a, \mathsf{sel}) \mapsto b] \# \mathit{split}$, no such local node from $\mathsf{s}[(a, \mathsf{sel}) \mapsto b]$ that is reachable through b points outside $\mathsf{s}[(a, \mathsf{sel}) \mapsto b]$. Hence $(\mathsf{s} * \mathsf{s}')[(a, \mathsf{psel}) \mapsto b] \# \mathit{split}$.

Splitting will add a new flow graph fg' to the shared heap of $(\mathsf{s} * \mathsf{s}')[(a, \mathsf{psel}) \mapsto b]$. The result is

$$\begin{aligned} & (\mathsf{s} * \mathsf{s}')[(a, \mathsf{psel}) \mapsto b].\mathsf{g}.\mathit{fg} * \mathit{fg}' \\ \overset{(1)}{=}\; & (\mathsf{s}[(a, \mathsf{psel}) \mapsto b] \times \mathsf{s}').\mathsf{g}.\mathit{fg} * \mathit{fg}' \\ \overset{(2)}{=}\; & \mathsf{s}[(a, \mathsf{psel}) \mapsto b].\mathsf{g}.\mathit{fg} * \mathsf{s}'.\mathsf{g}.\mathit{fg} * \mathit{fg}' \\ \overset{(3)}{=}\; & (\mathsf{s}[(a, \mathsf{psel}) \mapsto b].\mathsf{g}.\mathit{fg} * \mathit{fg}') * \mathsf{s}'.\mathsf{g}.\mathit{fg}. \end{aligned}$$

where the qualities follow from (1) the above discussion, (2) componentwise separating conjunction, and (3) commutativity. This is the shared heap of $\mathit{split}(\mathsf{s}[(a, \mathsf{sel} \mapsto b])$ composed with the shared heap of s'. For the local heap, we already discussed above that we only move nodes from $\mathsf{s.l.}\mathit{fg}$. Hence, s' remains unchanged. □

For modifications of program variables, there is a related result. Since stacks are not split, however, the modification applies to both states of the separating conjunction.

Lemma 36. *Consider $\mathsf{s} \# \mathsf{s}'$ and $px \in \mathit{GPVar} \Rightarrow b \in \mathsf{s.}\mathit{fg}.X$ and $\mathsf{s}[px \mapsto b] \# \mathit{split}$. Then we have $(\mathsf{s} * \mathsf{s}')[px \mapsto b] \# \mathit{split}$ and $\mathit{split}((\mathsf{s} * \mathsf{s}')[px \mapsto b]) = \mathit{split}(\mathsf{s}[px \mapsto b]) * \mathsf{s}'[px \mapsto b]$. The latter separating conjunction is defined.*

Proof. Assume $px \in \mathit{GPVar}$, otherwise there is no splitting. We have

$$(\mathsf{s} * \mathsf{s}')[px \mapsto b] = \mathsf{s}[px \mapsto b] * \mathsf{s}'[px \mapsto b]$$

since stacks are not split. The claim then follows with an argumentation similar to the one in the previous lemma. □

We also have a modification that corresponds to an allocation. For defining allocations, we introduce the *initial flow graph* for adress $a \in \mathbb{N}$, which is

$$\mathit{initfg}(a) \quad \triangleq \quad (\{a\}, \{\, (a, \mathsf{sel}, a) \;\mid\; \mathsf{sel} \in \mathit{Sel}\,\}, \{\, (b, \lambda_0, a) \;\mid\; b \in \mathbb{N} \setminus \{a\}\,\})\ .$$

In the latter set, λ_0 is the constant 0 function. Every selector points to the fresh node and the inflow is 0 on all incoming edges. Allocation is then captured by the function

$$\mathit{ext} : \Sigma \times \mathbb{N} \rightharpoonup \Sigma.$$

It extends the local flow graph by an initial flow graph for the given address. The function yields

$$\mathit{ext}(\mathsf{s}, a) \triangleq (\mathsf{s.g}, (\mathsf{s.l.}\mathit{fg} * \mathit{initfg}(a), \mathsf{s.l.}\mathit{val}))\ .$$

It is defined only if the address is fresh, $a \notin \mathsf{s.}\mathit{fg}.X$, and not pointed to from a global variable or the global heap of s. We denote this definedness by $(\mathsf{s}, a) \# \mathit{ext}$. The result is well-split.

$$
\begin{array}{rcl}
[\![\mathtt{skip}]\!](\mathrm{s}) & \triangleq & \{\, \mathrm{s} \,\} \\
[\![px := \mathtt{malloc}]\!](\mathrm{s}) & \triangleq & \{\, \mathit{split}(\mathrm{s}'[px \mapsto a]) \ \mid\ a \notin \mathrm{s}.\mathit{fg}.X \wedge \mathrm{s}' = \mathit{ext}(\mathrm{s}, a) \,\} \\
[\![\mathtt{assume}\ \mathrm{pred}(px_1, \dots, px_n)]\!](\mathrm{s}) & \triangleq & [\![\mathrm{pred}(px_1, \dots, px_n)]\!]\ \mathrm{s}.\mathit{val}\ ?\ \{\, \mathrm{s} \,\}\ :\ \varnothing \\
[\![\mathtt{assert}\ \mathrm{pred}(px_1, \dots, px_n)]\!](\mathrm{s}) & \triangleq & [\![\mathrm{pred}(px_1, \dots, px_n)]\!]\ \mathrm{s}.\mathit{val} \triangleright \{\, \mathrm{s} \,\} \\
[\![px := \mathrm{aop}(px_1, \dots, px_n)]\!](\mathrm{s}) & \triangleq & \mathit{assignfromop} \triangleright \{\, \mathit{split}(\mathrm{s}[px \mapsto b]) \,\} \\
[\![px := qx \rightarrow \mathrm{sel}]\!](\mathrm{s}) & \triangleq & \mathit{assignfromsel} \triangleright \{\, \mathit{split}(\mathrm{s}[px \mapsto b]) \,\} \\
[\![px \rightarrow \mathrm{sel} := qx]\!](\mathrm{s}) & \triangleq & \mathit{assigntosel} \triangleright \{\, \mathit{split}(\mathrm{s}[(a, \mathrm{sel}) \mapsto b]) \,\}.
\end{array}
$$

Fig. 12. Semantics of commands.

Lemma 37. *If* $(\mathrm{s}, a) \# \mathit{ext}$ *then* $\mathit{ext}(\mathrm{s}, a) \in \Sigma$.

D.2 Commands

The set of **commands** is

$$
\begin{array}{rl}
\mathtt{com} ::= & \mathtt{skip} \ \mid\ px := \mathrm{aop}(qx_1, \dots, qx_n) \ \mid\ px := qx \rightarrow \mathrm{sel} \ \mid\ px \rightarrow \mathrm{sel} := qx \\
& \mid\ px := \mathtt{malloc} \ \mid\ \mathtt{assume}\ \mathrm{pred}(px_1, \dots, px_n) \ \mid\ \mathtt{assert}\ \mathrm{pred}(px_1, \dots, px_n)\ .
\end{array}
$$

Here, $px, qx \in \mathit{PVar}$ are program variables, $\mathrm{sel} \in \mathit{Sel}$ is a selector, and $\mathrm{aop} : \mathbb{N}^n \to \mathbb{N}$ and $\mathrm{pred} : \mathbb{N}^n \to \mathbb{B}$ are an operation resp. a predicate over the address domain.

The semantics is strict in abort:

$$[\![\mathtt{com}]\!](\mathrm{abort}) \ \triangleq\ \{\, \mathrm{abort} \,\}\ .$$

For a state $\mathrm{s} \neq \mathrm{abort}$, we define the following preconditions:

$$
\begin{array}{rcl}
\mathit{assignfromop} & \triangleq & b = [\![\mathrm{aop}(px_1, \dots, px_n)]\!]\ \mathrm{s}.\mathit{val} \\
& & \wedge\ (px \in \mathit{GPVar} \Rightarrow b \in \mathrm{s}.\mathit{fg}.X) \wedge \mathrm{s}[px \mapsto b] \# \mathit{split} \\
\mathit{assignfromsel} & \triangleq & a = [\![qx]\!]\ \mathrm{s}.\mathit{val} \wedge a \in \mathrm{s}.\mathit{fg}.X \wedge b = \mathrm{s}.\mathit{val}(a, \mathrm{sel}) \\
& & \wedge\ (px \in \mathit{GPVar} \Rightarrow b \in \mathrm{s}.\mathit{fg}.X) \wedge \mathrm{s}[px \mapsto b] \# \mathit{split} \\
\mathit{assigntosel} & \triangleq & a = [\![px]\!]\ \mathrm{s}.\mathit{val} \wedge b = [\![qx]\!]\ \mathrm{s}.\mathit{val} \wedge \mathrm{s} \# [(a, \mathrm{sel}) \mapsto b] \\
& & \wedge\ (a \in \mathrm{s}.\mathit{fg}.\mathrm{g}.X \Rightarrow b \in \mathrm{s}.\mathit{fg}.X) \wedge \mathrm{s}[(a, \mathrm{sel}) \mapsto b] \# \mathit{split}.
\end{array}
$$

We use the notation *condition* $\triangleright$ *value* as a short hand for the expression *condition* ? *value* : $\{\mathrm{abort}\}$. The semantics of commands is given in Figure 12.

D.3 Assertions

We use countable sets *ALVar* and *FLVar* of logical variables with typical elements *ax* resp. *fx*. The first set holds variables that store addresses, the second holds variables for flow values. The logical variables are assumed to be disjoint and disjoint from the program variables. We use

$$\mathit{LogVal} \triangleq (\mathit{ALVar} \to \mathbb{N}) \uplus (\mathit{FLVar} \to \mathbb{M})$$

for the set of all valuations of the logical variables that respect the typing. We write

$$\mathit{ProgLogVal} \triangleq \mathit{Val}(\mathit{LPVar}) \uplus \mathit{Val}(\mathit{GPVar}) \uplus \mathit{LogVal}$$

for all valuations of the program and logical variables.

Terms are either *address terms* of type $\mathbb{N}$ or *flow terms* of type $\mathbb{M}$. We write $\mathbb{D}$ to mean either $\mathbb{N}$ or $\mathbb{M}$. To form terms, we assume the address domain and the flow domain each come with a set of

$$
\begin{array}{lll}
fg, plv \models \mathsf{emp} & \text{if} & fg.X = \varnothing \\
fg, plv \models \mathsf{pred}(term_1, \dots, term_n) & \text{if} & \mathsf{pred}(\llbracket term_1 \rrbracket\, plv, \dots, \llbracket term_n \rrbracket\, plv) \\
fg, plv \models aterm_1 \mapsto_{\mathsf{sel}} aterm_2 & \text{if} & \exists a, b.\ a = \llbracket aterm_1 \rrbracket\, plv \ \wedge\ b = \llbracket aterm_2 \rrbracket\, plv \\
& & \qquad \wedge\ fg.X = \{a\} \ \wedge\ fg.sval(a, \mathsf{sel}) = b \\
fg, plv \models aterm \mapsto_{\mathsf{in}} fterm & \text{if} & \exists a, m.\ a = \llbracket aterm \rrbracket\, plv \ \wedge\ m = \llbracket fterm \rrbracket\, plv \\
& & \qquad \wedge\ fg.X = \{a\} \ \wedge\ fg.in(a) = m \\
fg, plv \models aterm \mapsto_{\mathsf{flow}} fterm & \text{if} & \exists a, m.\ a = \llbracket aterm \rrbracket\, plv \ \wedge\ m = \llbracket fterm \rrbracket\, plv \\
& & \qquad \wedge\ fg.X = \{a\} \ \wedge\ fg.flow(a) = m.
\end{array}
$$

Fig. 13. Satisfaction relation for atomic predicates.

operations *AOP* resp. *FOP* defined on them. Each operation has a type. For address operations, this type has the form $\mathbb{D}_1 \to \dots \to \mathbb{D}_n \to \mathbb{N}$. For flow operations, the result is a monoid value. To ease the notation, we simplify the types to $(\mathbb{N} \uplus \mathbb{M})^n \to \mathbb{D}$ and write *term* for an address or a flow term. In applications, we will respect the precise types. We also assume to have a set of predicates *Pred* that follow a similar typing scheme.

The sets of **address** and **flow terms** are defined by mutual induction:

$$
\begin{array}{lcl}
aterm & ::= & a \ \mid\ ax \ \mid\ px \ \mid\ \mathsf{aop}(term_1, \dots, term_n) \\
fterm & ::= & m \ \mid\ fx \ \mid\ \mathsf{fop}(term_1, \dots, term_n).
\end{array}
$$

An address term is an address $a \in \mathbb{N}$, a logical address variable $ax \in ALVar$, a program variable $px \in PVar$, and an operation $\mathsf{aop} \in AOP$ applied to n terms, with n is the operation's arity. A flow term is a flow value $m \in \mathbb{M}$, a logical flow variable $fx \in FLVar$, and a flow operation $\mathsf{fop} \in FOP$ applied to terms.

The semantics expects a valuation *plv* of the program and logical variables and returns an address or flow value:

$$\llbracket term \rrbracket : ProgLogVal \to \mathbb{N} \uplus \mathbb{M}.$$

The definition is as follows:

$$
\llbracket d \rrbracket\, plv \ \triangleq\ d \qquad\qquad \llbracket x \rrbracket\, plv \ \triangleq\ plv(x)
$$
$$
\llbracket \mathsf{op}(term_1, \dots, term_n) \rrbracket\, plv \ \triangleq\ \mathsf{op}(\llbracket term_1 \rrbracket\, plv, \dots, \llbracket term_n \rrbracket\, plv)\ .
$$

Here, d is an address $a \in \mathbb{N}$ or a flow value $m \in \mathbb{M}$, x is a logical flow variable $fx \in FLVar$, a logical address variable $ax \in ALVar$, or a program variable $px \in PVar$, and op is a flow operation $\mathsf{fop} \in FOP$ or an address operation $\mathsf{aop} \in AOP$.

The set of **atomic predicates** is defined as follows, with $\mathsf{sel} \in Sel$:

$$
\begin{array}{rl}
apred \ ::= & \mathsf{emp} \ \mid\ \mathsf{pred}(term_1, \dots, term_n) \\
& \mid\ aterm_1 \mapsto_{\mathsf{sel}} aterm_2 \ \mid\ aterm \mapsto_{\mathsf{in}} fterm \ \mid\ aterm \mapsto_{\mathsf{flow}} fterm.
\end{array}
$$

A flow graph *fg* and a valuation *plv* of the program and logical variables satisfy an atomic predicate as prescribed by Figure 13.

The set of **state predicates** is defined by

$$
A \quad ::= \quad apred \ \mid\ \neg A \ \mid\ A_1 \wedge A_2 \ \mid\ \exists ax.A \ \mid\ \exists fx.A \ \mid\ A_1 * A_2 \ \mid\ A_1 \mathrel{-\!\!*} A_2 \ \mid\ \boxed{A}\ .
$$

It will be convenient to have the box predicate $\boxed{A}$ as introduced in RGSep [Vafeiadis 2008; Vafeiadis and Parkinson 2007]. The predicate is meant to describe constraints over the shared state. Therefore, we expect A to contain neither local program variables nor further boxes.

The abort state never satisfies a state predicate.

For states $\mathsf{s} = (\mathsf{g}, \mathsf{l}) \in \Sigma$ and valuations lv of the logical variables that occur free in the formula the satisfaction relation is defined as follows:

$$
\begin{array}{lll}
\mathsf{s}, lv \models apred & \text{if} & \mathsf{s.l}.fg, lv \uplus \mathsf{s}.val \models apred \\
\mathsf{s}, lv \models \neg A & \text{if} & \mathsf{s}, lv \not\models A \\
\mathsf{s}, lv \models A_1 \land A_2 & \text{if} & \mathsf{s}, lv \models A_1 \quad \land \quad \mathsf{s}, lv \models A_2 \\
\mathsf{s}, lv \models \exists ax.A & \text{if} & \exists a \in \mathbb{N}.\ \ \mathsf{s}, lv[ax \mapsto a] \models A \\
\mathsf{s}, lv \models \exists fx.A & \text{if} & \exists m \in \mathbb{M}.\ \ \mathsf{s}, lv[fx \mapsto m] \models A \\
\mathsf{s}, lv \models A_1 * A_2 & \text{if} & \exists \mathsf{l}_1, \mathsf{l}_2.\ \ \mathsf{l}_1 \# \mathsf{l}_2 \ \land\ \mathsf{l}_1 * \mathsf{l}_2 = \mathsf{s.l} \ \land (\mathsf{s.g}, \mathsf{l}_1), lv \models A_1 \ \land\ (\mathsf{s.g}, \mathsf{l}_2), lv \models A_2 \\
\mathsf{s}, lv \models A_1 \mathrel{-\!\!*} A_2 & \text{if} & \forall \mathsf{s}'.\ \ (\mathsf{s} \# \mathsf{s}' \ \land\ \mathsf{s}', lv \models A_1) \Rightarrow\ \mathsf{s} * \mathsf{s}', lv \models A_2 \\
\mathsf{s}, lv \models \boxed{A} & \text{if} & \mathsf{s.l} \in \mathsf{emp}_{\mathsf{L}} \ \ \land\ \ \mathsf{s.g}, lv \models' A.
\end{array}
$$

The definition of $\models'$ is similar. The key difference is that we split the global flow graph when defining the separating conjunction, and similar for the separating implication. Note that by the assumption on the shape of formulas, we do not need a valuation of the local variables to determine the truth of atomic predicates that occur inside boxes.

The set of **computation predicates** is

$$C \quad ::= \quad \_A \ \mid\ \Diamond A \ \mid\ C \land C \ \mid\ C \lor C \ \mid\ C * C \ \mid\ \exists ax.C \ \mid\ \exists fx.C.$$

The semantics is defined over non-empty sequences σ with $\sigma \in (\Sigma \uplus \{\text{abort}\})^+$ and valuations of logical variables lv as follows. A sequence containing abort never satisfies a predicate. For a sequence $\sigma \in \Sigma^+$, we define:

$$
\begin{array}{lll}
\sigma.\mathsf{s}, lv \models \_A & \text{if} & \mathsf{s}, lv \models A \\
\sigma, lv \models \Diamond A & \text{if} & \exists \mathsf{s} \in \Sigma, \sigma_1, \sigma_2 \in \Sigma^*. \sigma = \sigma_1.\mathsf{s}.\sigma_2 \land\ \mathsf{s}, lv \models A \\
\sigma, lv \models C_1 \land C_2 & \text{if} & \sigma, lv \models C_1 \quad \land \quad \sigma, lv \models C_2 \\
\sigma, lv \models C_1 \lor C_2 & \text{if} & \sigma, lv \models C_1 \quad \lor \quad \sigma, lv \models C_2 \\
\sigma, lv \models C_1 * C_2 & \text{if} & \exists \sigma_1, \sigma_2 \in \Sigma^+.\ \ \sigma_1 \# \sigma_2 \ \land\ \sigma = \sigma_1 * \sigma_2 \ \land\ \sigma_1, lv \models C_1 \ \land\ \sigma_2, lv \models C_2 \\
\sigma, lv \models \exists ax.C & \text{if} & \exists a \in \mathbb{N}.\ \ \sigma, lv[ax \mapsto a] \models C \\
\sigma, lv \models \exists fx.C & \text{if} & \exists m \in \mathbb{M}.\ \ \sigma, lv[fx \mapsto m] \models C.
\end{array}
$$

A computation predicate is *closed* if it does not contain free logical variables. In this case, the satisfaction relation only depends on the computation but is independent of the valuation. We write $[\![C]\!]$ for the set of computations that satisfy a predicate. We need the following lemma, which is a consequence of Lemma 13 with the consideration of quantifiers added.

Lemma 38. *$[\![C]\!]$ is frameable for every C.*

Lemma 39. *If we have $\sigma.\mathsf{s} \in [\![C]\!]$ and $px \notin \mathrm{fv}(C)$, then we also have $\sigma.\mathsf{s}.(\mathsf{s}[px \mapsto a]) \in [\![C]\!]$.*

In addition to the equivalences listed in Lemma 14, we have the following for quantifiers over logical variables.

Lemma 40. *We have:*

$$\sigma, lv \models \exists ax.\_A \iff \sigma, lv \models \_(\exists ax.A) \qquad \text{and} \qquad \sigma, lv \models \exists fx.\_A \iff \sigma, lv \models \_(\exists fx.A)\ .$$

D.4 Semantics

With the development in Section 7, the semantics generalizes to computations. The key property required in Section 3 is as follows.

Lemma 41. *Let* com *be different from* $px := \mathrm{aop}(qx_1, \ldots, qx_n)$, $px := qx \to \mathsf{sel}$, *and* $px := \texttt{malloc}$. *Then* $[\![\![\mathsf{com}]\!]\!]([\![C_1]\!]) \subseteq [\![C_2]\!]$ *implies* $[\![\![\mathsf{com}]\!]\!]([\![C_1 * C]\!]) \subseteq [\![C_2 * C]\!]$.

Proof. Recall: $[\![C_1 * C]\!]$ contains sequences of states $\sigma.\mathrm{s} \in \Sigma^+$. It is readily checked that $[\![C_1 * C]\!] = [\![C_1]\!] * [\![C]\!]$. So there are $\sigma_1.\mathrm{s}_1 \in [\![C_1]\!]$ and $\sigma_2.\mathrm{s}_2 \in [\![C_2]\!]$ with $\sigma.\mathrm{s} = \sigma_1.\mathrm{s}_1 * \sigma_2.\mathrm{s}_2$. We have that $[\![\![\mathsf{com}]\!]\!](\sigma.\mathrm{s})$ extends the sequence $\sigma.\mathrm{s}$ by elements from $[\![\mathsf{com}]\!](\mathrm{s})$. Such elements are either abort or states.

Case abort. We show that $[\![\mathsf{com}]\!](\mathrm{s}) = \mathrm{abort}$ implies $[\![\mathsf{com}]\!](\mathrm{s}_1) = \mathrm{abort}$. Then $[\![\![\mathsf{com}]\!]\!](C_1)$ contains a sequence with abort, contradicting $[\![\![\mathsf{com}]\!]\!]([\![C_1]\!]) \subseteq [\![C_2]\!]$. We distinguish the following cases:

- Skip and assume do not abort.
- An assertion only depends on the valuation of the program variables. Valuations are never split. This means s_1 will have the same valuation, and lead to the same abort.
- We have excluded assignments to program variables and malloc.
- An assignment to a selector aborts.

It remains to argue for the last case. An assignment to a selector aborts if the condition *assigntosel* fails. We consider each case and argue that the abort also happens on s_1. Note that valuations are never split, so the addresses a and b are the same in s_1. If $\mathrm{s}\,\#[(a, \mathsf{sel}) \mapsto b]$ fails, we have $[\![up]\!](\mathrm{s}.c) = \mathrm{abort}$, with up induced by $[(a, \mathsf{sel}) \mapsto b]$. By Lemma 34 applied in contraposition, $[\![up]\!](\mathrm{s}_1.c) = \mathrm{abort}$. The contraposition applies because $\mathrm{s}_1 \# \mathrm{s}_2$ implies $\mathrm{s}_1.c \# \mathrm{s}_2.c$ and $\mathrm{s}_1.c * \mathrm{s}_2.c = \mathrm{s}.c$. Hence, the command will abort on s_1.

If the implication $a \in \mathrm{s}.fg.\mathrm{g}.X \Rightarrow b \in \mathrm{s}.fg.X$ fails, we have $a \in \mathrm{s}.fg.\mathrm{g}.X$ and $b \notin \mathrm{s}.fg.X$. To see that the implication also fails on s_1, consider first the case that the implication becomes trivial, $a \notin \mathrm{s}_1.fg.\mathrm{g}.X$. Then $[\![up]\!](\mathrm{s}_1.c) = \mathrm{abort}$, because the domain of the update induced by the modification is not included in the state. If the implication is non-trivial, $b \notin \mathrm{s}.fg.X$ implies $b \notin \mathrm{s}_1.fg.X$. This is because of $\mathrm{s}.fg.X = \mathrm{s}_1.fg.X \uplus \mathrm{s}_2.fg.X$. Hence the implication, and so the command, also fail on s_1.

If $\mathrm{s}[(a, \mathsf{sel}) \mapsto b]\,\#\,split$ does not hold, then Lemma 35 applies in contraposition and yields a failure of definedness, $\mathrm{s}_1[(a, \mathsf{sel}) \mapsto b]\,\#\,split$. A failure of $\mathrm{s}_1[(a, \mathsf{sel}) \mapsto b]\,\#\,split$ in turn means that the command aborts on s_1. To see that the contraposition applies, note that $\mathrm{s}_1 * \mathrm{s}_2 = \mathrm{s}$. Moreover, we can rely on the implication $a \in \mathrm{s}_1.fg.\mathrm{g}.X \Rightarrow b \in \mathrm{s}_1.fg.X$, for otherwise the command would abort on s_1. Similarly, if $\mathrm{s}_1\,\#[(a, \mathsf{sel}) \mapsto b]$ failed, we would also abort on s_1.

Case $\mathrm{s}' \in [\![\mathsf{com}]\!](\mathrm{s})$. We show that there is $\mathrm{s}_1' \in [\![\mathsf{com}]\!](\mathrm{s}_1)$ with $\mathrm{s}_1' \# \mathrm{s}_2$ and $\mathrm{s}' = \mathrm{s}_1' * \mathrm{s}_2$. Then $[\![\![\mathsf{com}]\!]\!]([\![C_1]\!]) \subseteq [\![C_2]\!]$, $\sigma_1.\mathrm{s}_1 \in [\![C_1]\!]$, $\mathrm{s}_1' \in [\![\mathsf{com}]\!](\mathrm{s}_1)$ imply $\sigma_1.\mathrm{s}_1.\mathrm{s}_1' \in [\![C_2]\!]$. Since $[\![C]\!]$ is frameable by Lemma 38, $\sigma_2.\mathrm{s}_2.\mathrm{s}_2 \in [\![C]\!]$. So, $\sigma.\mathrm{s}.\mathrm{s}' = \sigma_1.\mathrm{s}_1.\mathrm{s}_1' * \sigma_2.\mathrm{s}_2.\mathrm{s}_2 \in [\![C_2]\!] * [\![C]\!] = [\![C_2 * C]\!]$. Case analysis:

- For skip, we have $\mathrm{s}' = \mathrm{s}$ and $\mathrm{s}_1' = \mathrm{s}_1 \in [\![\texttt{skip}]\!](\mathrm{s}_1)$. Then $\mathrm{s}' = \mathrm{s}_1' * \mathrm{s}_2$.
- Assume and assert are similarly simple, as the semantics only depends on the program variable valuation and that is not split.
- We have excluded assignments and mallocs to program variables.
- For an assignment to a selector that results in the state $\mathrm{s}' = split(\mathrm{s}[(a, \mathsf{sel}) \mapsto b])$, we have condition *assigntosel* for s_1 as follows:

$$a = [\![px]\!]\ \mathrm{s}_1.val\ \wedge\ b = [\![qx]\!]\ \mathrm{s}_1.val\ \wedge \mathrm{s}_1\,\#[(a, \mathsf{sel}) \mapsto b]$$
$$\wedge\ \mathrm{s}_1[(a, \mathsf{sel}) \mapsto b]\,\#\,split\ \wedge (a \in \mathrm{s}_1.fg.\mathrm{g}.X \Rightarrow b \in \mathrm{s}_1.fg.X)\ .$$

Otherwise, the assignment would abort on s_1,contradicting $[\![\mathtt{com}]\!]([\![C_1]\!]) \subseteq [\![C_2]\!]$. When applied to s_1, we obtain $s_1' = \mathit{split}(s_1[(a,\mathsf{sel}) \mapsto b])$. Then

$$\begin{aligned} s_1' * s_2 &= \mathit{split}(s_1[(a,\mathsf{sel}) \mapsto b]) * s_2 \\ &= \mathit{split}((s_1 * s_2)[(a,\mathsf{sel}) \mapsto b]) = \mathit{split}(s[(a,\mathsf{sel}) \mapsto b]) = s' \end{aligned}$$

The first equality is the definition of s_1', the next is Lemma 35, the third is the fact that $s_1 * s_2 = s$, and the last is the definition of s'. The lemma applies by *assigntosel* stated above. This concludes the proof. □

Since we do not use program variables as resources, the above lemma will not hold without prerequisites for assignments to program variables and malloc. The prerequisite will be that the program variable modified by the command does not occur free in the predicate that is added with the separating conjunction. The prerequisite will then form a side-condition to applying the frame rule.

Lemma 42. *Let* com *be one of:* $px := \mathrm{aop}(qx_1, \ldots, qx_n)$ *or* $px := qx \rightarrow \mathsf{sel}$ *or* $px := \mathtt{malloc}$. *Let* $px \notin \mathrm{fv}(C)$. *Then we have:* $[\![\mathtt{com}]\!]([\![C_1]\!]) \subseteq [\![C_2]\!]$ *implies* $[\![\mathtt{com}]\!]([\![C_1 * C]\!]) \subseteq [\![C_2 * C]\!]$.

Proof. The proof follows the same strategy as the proof for the previous lemma. Recall that $[\![C_1 * C]\!]$ contains sequences of states $\sigma.s \in \Sigma^+$. Hence, there are some $\sigma_1.s_1 \in [\![C_1]\!]$ and $\sigma_2.s_2 \in [\![C_2]\!]$ with $\sigma.s = \sigma_1.s_1 * \sigma_2.s_2$. Moreover, we have that $[\![\mathtt{com}]\!](\sigma.s)$ extends the sequence $\sigma.s$ by elements from $[\![\mathtt{com}]\!](s)$. Such elements are either abort or states.

Case abort. We are going to show that $[\![\mathtt{com}]\!](s) = \mathrm{abort}$ implies $[\![\mathtt{com}]\!](s_1) = \mathrm{abort}$. Then $[\![\mathtt{com}]\!](C_1)$ contains a sequence with abort, contradicting $[\![\mathtt{com}]\!]([\![C_1]\!]) \subseteq [\![C_2]\!]$.

- Consider $px := \mathrm{aop}(qx_1, \ldots, qx_n)$. It aborts due to a failure of the precondition *assignfromop*. The expression will evaluate to the same address b in s_1, because stacks are never split. Assume the implication fails, meaning $px \in \mathit{GPVar}$ and $b \notin s.\mathit{fg}.X$. Then $b \notin s_1.\mathit{fg}.X$ as $s.\mathit{fg}.X = s_1.\mathit{fg}.X \uplus s_2.\mathit{fg}.X$. Hence, the implication will also fail on s_1 and we have the desired abort.

  If $s[px \mapsto b] \# \mathit{split}$ fails, then either the local and the global flow graph of $s[px \mapsto b]$ do not compose or we reach a local node from the global state that points to a node outside the state. In the former case, we note that $s[px \mapsto b].\mathsf{g}.\mathit{fg} = s.\mathsf{g}.\mathit{fg}$ and $s[px \mapsto b].\mathsf{l}.\mathit{fg} = s.\mathsf{l}.\mathit{fg}$. Moreover, we know that $s.\mathsf{g}.\mathit{fg} \# s\mathsf{l}.\mathit{fg}$ as states are well-split. The case does not occur.

  In the latter case, the node that points outside s is reachable through b. To see this, note that s is well-split and hence there is no way for the global state to reach the local heap. In this case, split is defined (and the identity). Hence, the only way to make split fail is by following the pointer from px to b and potentially further to the mentioned node. In $s_1[px \mapsto b]$, we may be able to follow the same path, in which case split would also fail on $s_1[px \mapsto b]$ due to the same node. Alternatively, the path may lead outside $s_1[px \mapsto b]$. In that case, there is a first node on the path that leaves $s_1[px \mapsto b]$. This first node points outside $s_1[px \mapsto b]$, namely to the next node on the path. As a result, split also fails on $s_1[px \mapsto b]$, as required.
- Consider $px := qx \rightarrow \mathsf{sel}$. The command aborts due to a failure of the precondition *assignfromsel* The command may abort as $a \notin s.\mathit{fg}.X$. Then $a \notin s_1.\mathit{fg}.X$ by $s_1.\mathit{fg}.X \uplus s_2.\mathit{fg}.X = s.\mathit{fg}.X$. Hence, the command will also abort on s_1.

  For the implication and splitting, the argumentation is like in the previous case.
- A malloc does not abort, so the case does not occur.

Case $s' \in [\![\mathtt{com}]\!](s)$. We show there are $s_1' \in [\![\mathtt{com}]\!](s_1)$ and $\sigma_2.s_2.s_2' \in [\![C]\!]$ with $s_1' \# s_2'$ and $s' = s_1' * s_2'$. From the former condition and the assumption, we get $\sigma_1.s_1.s_1' \in [\![C_2]\!]$. Hence, $\sigma.s.s' = \sigma_1.s_1.s_1' * \sigma_2.s_2.s_2' \in [\![C_2]\!] * [\![C]\!] = [\![C_2 * C]\!]$. We distinguish the cases.

- Consider $px := \mathrm{aop}(qx_1, \ldots, qx_n)$. The resulting state is $s' = \mathit{split}(s[px \mapsto b])$ with
$$b = [\![\mathrm{aop}(px_1, \ldots, px_n)]\!]\ s.\mathit{val}\ .$$
The command does not abort on s_1, as this would contradict $[\![\mathtt{com}]\!]([\![C_1]\!]) \subseteq [\![C_2]\!]$. Moreover, the valuations of the program variables will coincide for s and s_1. Hence, address b will not change and we get the precondition *assignfromop* for s_1:
$$b = [\![\mathrm{aop}(px_1, \ldots, px_n)]\!]\ s_1.\mathit{val}\ \wedge\ (px \in \mathit{GPVar} \Rightarrow b \in s_1.\mathit{fg}.X)\ \wedge\ s_1[px \mapsto b] \# \mathit{split}\ .$$
The state resulting from the command is
$$s_1' = \mathit{split}(s_1[px \mapsto b])\ .$$
Moreover, the precondition is strong enough to apply Lemma 36. With $s_2' = s_2[px \mapsto b]$, the lemma shows $s_1' \# s_2'$ and $s' = s_1' * s_2'$. For s_2', Lemma 39 yields $\sigma_2.s_2.s_2' \in [\![C]\!]$. This concludes the case.
- Consider $px := qx \to \mathrm{sel}$. The resulting state is this one: $s' = \mathit{split}(s[px \mapsto b])$ with $b = s.\mathit{sval}(a, \mathrm{sel})$ and $a = [\![qx]\!]\ s.\mathit{val}$. The command does not abort on s_1, as this would contradict $[\![\mathtt{com}]\!]([\![C_1]\!]) \subseteq [\![C_2]\!]$. Moreover, the valuations of the program variables will coincide for s and s_1. Hence, address a will not change and we get the precondition *assignfromsel* for s_1:
$$\begin{aligned} a = [\![qx]\!]\ s_1.\mathit{val}\ \wedge\ b' = s_1.\mathit{val}(a, \mathrm{sel})\ \wedge a \in s_1.\mathit{fg}.X \\ \wedge\ s_1[px \mapsto b'] \# \mathit{split}\ \wedge\ (px \in \mathit{GPVar} \Rightarrow b' \in s_1.\mathit{fg}.X)\ . \end{aligned}$$
Because $s = s_1 * s_2$ and $s_1.\mathit{val}(-)$ is a function, we have $b' = b$. Hence, the state resulting from the command is $s_1' = \mathit{split}(s_1[px \mapsto b])$. The remainder of the reasoning is as in the previous case.
- Consider $px := \mathtt{malloc}$. The resulting state is $s' = \mathit{split}(\tilde{s}[px \mapsto b])$ with $\tilde{s} = \mathit{ext}(s, b)$. Since $(s, b) \# \mathit{ext}$, we have $(s_1, b) \# \mathit{ext}$. The address is fresh for s_1 as $s_1.\mathit{fg}.X \uplus s_2.\mathit{fg}.X = s.\mathit{fg}.X$ and the address is fresh for s. To see that the global part of s_1 does not point to the address, note that the global part of s does not point to it and we have $s_1.g * s_2.g = s.g$. Hence, $(s_1, b) \# \mathit{ext}$ and we let $\tilde{s}_1 = \mathit{ext}(s_1, b)$. With the same argument, also the global part of s_2 does not point to the address. The global part of s_2 does not point to the remaining local heap of s_1 as $s_1 \# s_2$. With the same argument, the global part of s_1 does not point to the local heap of s_2. Hence, $\tilde{s}_1 \# s_2$. Since $s_1 * s_2 = s$, we moreover have $\tilde{s}_1 * s_2 = \tilde{s}$.

  Since s_1 is a state, it is well-split. By definition, b only points to itself. Hence, we have $\tilde{s}_1[px \mapsto b] \# \mathit{split}$ and set $s_1' = \mathit{split}(\tilde{s}_1[px \mapsto b])$. The splitting will move b to the shared heap should px be a shared variable. Lemma 36 applies and yields $s' = s_1' * s_2'$ where we have $s_2' = s_2[px \mapsto b]$. It is readily checked that $s_1' \in [\![px := \mathtt{malloc}]\!](s_1)$. Moreover, $\sigma_2.s_2.s_2' \in [\![C]\!]$ by Lemma 39.

This concludes the proof. □

E PROOF OF THEOREM 3: SOUNDNESS OF THE PROGRAM LOGIC

The guarantee given by the thread-local proofs constructed in our program logic is captured by the notion of safety. Safety guarantees that upon termination the postcondition holds, $\mathrm{acc}(\mathtt{st}, r, p)$ is defined as $\mathtt{st} = \mathtt{skip} \Rightarrow r \subseteq p$. Moreover, for every command to be executed we are sure to have captured the interference and to execute safely for another k steps from the resulting predicate.

Definition 43. *We define* $\mathsf{safe}^0_{\mathbb{P},\mathbb{I}}(\mathtt{st}, r, p) \triangleq \mathit{true}$ *and* $\mathsf{safe}^{k+1}_{\mathbb{P},\mathbb{I}}(\mathtt{st}, r, p) \triangleq (1) \wedge (2)$ *with*

(1) $\mathsf{acc}(\mathtt{st}, r, p)$

(2) $\forall \mathtt{com}.\forall \mathtt{st}'.\ \mathtt{st} \xrightarrow{\mathtt{com}} \mathtt{st}' \ \Rightarrow\ \mathsf{inter}(r, \mathtt{com}) \subseteq \mathbb{I} \ \wedge\ \exists n \in \mathbb{P}.\ [\![\mathtt{com}]\!](r) \subseteq n \ \wedge\ \mathsf{safe}^k_{\mathbb{P},\mathbb{I}}(\mathtt{st}', n, p)$.

The predicate is monotonic in the various arguments as follows.

Lemma 44. *Consider* $\mathbb{P}_1 \subseteq \mathbb{P}_2$, $\mathbb{I}_1 \subseteq \mathbb{I}_2$, $p_1 \supseteq p_2$, $q_1 \subseteq q_2$, *and* $k_1 \geq k_2$. *Then* $\mathsf{safe}^{k_1}_{\mathbb{P}_1,\mathbb{I}_1}(\mathtt{st}, p_1, q_1)$ *implies* $\mathsf{safe}^{k_2}_{\mathbb{P}_2,\mathbb{I}_2}(\mathtt{st}, p_2, q_2)$.

Proof. Consider $\mathbb{P}_1 \subseteq \mathbb{P}_2$, $\mathbb{I}_1 \subseteq \mathbb{I}_2$, and $q_1 \subseteq q_2$.
We proceed by induction on k_1.
$k_1 = 0$: There is nothing to do.
$k_1 + 1$: The induction hypothesis is

$$\forall k_2.\forall \mathtt{st}.\forall p_1.\forall p_2.\quad k_2 \leq k_1 \wedge p_1 \supseteq p_2 \wedge \mathsf{safe}^{k_1}_{\mathbb{P}_1,\mathbb{I}_1}(\mathtt{st}, p_1, q_1) \implies \mathsf{safe}^{k_2}_{\mathbb{P}_2,\mathbb{I}_2}(\mathtt{st}, p_2, q_2)\ .$$

Further, consider $\mathtt{st}$ and p_1 so that $\mathsf{safe}^{k_1+1}_{\mathbb{P}_1,\mathbb{I}_1}(\mathtt{st}, p_1, q_1)$. Consider $k_2 \leq k_1 + 1$ and $p_2 \subseteq p_1$. We show

$$\mathsf{safe}^{k_2}_{\mathbb{P}_2,\mathbb{I}_2}(\mathtt{st}, p_2, q_2)\ .$$

We can assume $0 < k_2$, otherwise there is nothing to do.
(1) We have $\mathsf{safe}^{k_1+1}_{\mathbb{P}_1,\mathbb{I}_1}(\mathtt{st}, p_1, q_1)$. Hence, if $\mathtt{st} = \mathtt{skip}$, then $p_1 \subseteq q_1$. We have $p_2 \subseteq p_1$ and $q_1 \subseteq q_2$ by assumption. Hence, $p_2 \subseteq q_2$ as required.
(2a) Consider $\mathtt{st} \xrightarrow{\mathtt{com}} \mathtt{st}'$. We have $\mathsf{safe}^{k_1+1}_{\mathbb{P}_1,\mathbb{I}_1}(\mathtt{st}, p_1, q_1)$. Hence, $\mathsf{inter}(p_1, \mathtt{com}) \subseteq \mathbb{I}_1$. This means there is $(o, \mathtt{com}) \in \mathbb{I}_1$ with $p_1 \subseteq o$. We have $p_2 \subseteq p_1$ and so $p_2 \subseteq o$. Hence, $\mathsf{inter}(p_2, \mathtt{com}) \subseteq \mathbb{I}_1$. We have $\mathbb{I}_1 \subseteq \mathbb{I}_2$ by assumption. Hence, $\mathsf{inter}(p_2, \mathtt{com}) \subseteq \mathbb{I}_2$ as required.
(2b) Because of $\mathsf{safe}^{k_1+1}_{\mathbb{P}_1,\mathbb{I}_1}(\mathtt{st}, p_1, q_1)$, there is $n \in \mathbb{P}_1$ as follows. We have $[\![\mathtt{com}]\!](p_1) \subseteq n$ as well as $\mathsf{safe}^{k_1}_{\mathbb{P}_1,\mathbb{I}_1}(\mathtt{st}', n, q_1)$. As $\mathbb{P}_1 \subseteq \mathbb{P}_2$, we also have $n \in \mathbb{P}_2$. We moreover have $p_2 \subseteq p_1$ and hence $[\![\mathtt{com}]\!](p_2) \subseteq [\![\mathtt{com}]\!](p_1)$. We conclude $[\![\mathtt{com}]\!](p_2) \subseteq n$ as required.
(2c) It remains to argue for $\mathsf{safe}^{k_2-1}_{\mathbb{P}_2,\mathbb{I}_2}(\mathtt{st}', n, q_2)$. We have $0 < k_2 \leq k_1 + 1$. Hence, $0 \leq k_2 - 1 \leq k_1$. We have $\mathsf{safe}^{k_1}_{\mathbb{P}_1,\mathbb{I}_1}(\mathtt{st}', n, q_1)$. The induction hypothesis yields $\mathsf{safe}^{k_2-1}_{\mathbb{P}_2,\mathbb{I}_2}(\mathtt{st}', n, q_2)$. □

Soundness of the thread-local derivation is stated in the next proposition.

Proposition 45. *Consider* $\mathbb{P}, \mathbb{I} \Vdash \{\, p \,\}\ \mathtt{st}\ \{\, q \,\}$ *and* $k \in \mathbb{N}$. *We have* $\mathsf{safe}^k_{\mathbb{P},\mathbb{I}}(\mathtt{st}, p, q)$.

The proof of Proposition 45 proceeds by rule induction over the derivation rules of the program logic (Figure 5). We break the proof down into individual lemmas based on the case analysis of the last derivation rule used in the proof.

E.1 Soundness of SEQ

Lemma 46. *If* $\mathsf{safe}^k_{\mathbb{P}_1,\mathbb{I}_1}(\mathtt{st}_1, p, q)$ *and* $\mathsf{safe}^k_{\mathbb{P}_2,\mathbb{I}_2}(\mathtt{st}_2, q, o)$ *and* $\mathbb{P} = \mathbb{P}_1 \cup \mathbb{P}_2$ *and* $\mathbb{I} = \mathbb{I}_1 \cup \mathbb{I}_2$, *then* $\mathsf{safe}^k_{\{q\}\cup\mathbb{P},\mathbb{I}}(\mathtt{st}_1; \mathtt{st}_2, p, o)$.

Proof. Consider $\mathtt{st}_2$ and q. We proceed by induction on k.
$k = 0$: Trivial.
$k + 1$: The induction hypothesis is

$$\forall \mathtt{st}_1.\forall p.\quad \mathsf{safe}^k_{\mathbb{P}_1,\mathbb{I}_1}(\mathtt{st}_1, p, q) \wedge \mathsf{safe}^k_{\mathbb{P}_2,\mathbb{I}_2}(\mathtt{st}_2, q, o) \implies \mathsf{safe}^k_{\{q\}\cup\mathbb{P},\mathbb{I}}(\mathtt{st}_1; \mathtt{st}_2, p, o)\ .$$

Consider $\mathtt{st}_1$ and p so that

$$\mathsf{safe}^{k+1}_{\mathbb{P}_1,\mathbb{I}_1}(\mathtt{st}_1, p, q) \wedge \mathsf{safe}^{k+1}_{\mathbb{P}_2,\mathbb{I}_2}(\mathtt{st}_2, q, o)\ .$$

Our goal is to show

$$\mathsf{safe}^{k+1}_{\{q\}\cup\mathbb{P},\mathbb{I}}(\mathtt{st}_1;\mathtt{st}_2, p, o)\ .$$

(1) As $\mathtt{st}_1;\mathtt{st}_2 \neq \mathtt{skip}$, there is nothing to show.
(2) Consider $\mathtt{com}$ and $\mathtt{st}'$ with $\mathtt{st}_1;\mathtt{st}_2 \xrightarrow{\mathtt{com}} \mathtt{st}'$.
There are two cases for transitions from $\mathtt{st}_1;\mathtt{st}_2$.

Case 1: $\mathtt{st}_1 = \mathtt{skip}$ and $\mathtt{skip};\mathtt{st}_2 \xrightarrow{\mathtt{skip}} \mathtt{st}_2$.
(2a) We have $\neg\mathsf{eff}(p, \mathtt{skip})$, hence $\mathsf{inter}(p, \mathtt{skip}) = \varnothing \subseteq \mathbb{I}$.
(2b) We argue that q is the right choice for a predicate. It is in $\{q\} \cup \mathbb{P}$. We have $\mathsf{safe}^{k+1}_{\mathbb{P}_1,\mathbb{I}_1}(\mathtt{st}_1, p, q)$ by assumption and $\mathtt{st}_1 = \mathtt{skip}$. Hence, $p \subseteq q$ holds. With this, $[\![\mathtt{skip}]\!](p) = p \subseteq q$ follows.
(2c) By assumption, $\mathsf{safe}^{k+1}_{\mathbb{P}_2,\mathbb{I}_2}(\mathtt{st}_2, q, o)$. Monotonicity in Lemma 44 then yields $\mathsf{safe}^{k}_{\{q\}\cup\mathbb{P},\mathbb{I}}(\mathtt{st}_2, q, o)$.

Case 2: $\mathtt{st}_1 \xrightarrow{\mathtt{com}} \mathtt{st}_1'$ and $\mathtt{st}_1;\mathtt{st}_2 \xrightarrow{\mathtt{com}} \mathtt{st}_1';\mathtt{st}_2$.
(2a) By the assumption of the induction step, we have the following: $\mathsf{safe}^{k+1}_{\mathbb{P}_1,\mathbb{I}_1}(\mathtt{st}_1, p, q)$. Hence, $\mathsf{inter}(p, \mathtt{com}) \subseteq \mathbb{I}_1 \subseteq \mathbb{I}$.
(2b) Furthermore, there is some $n \in \mathbb{P}_1$ with $[\![\mathtt{com}]\!](p) \subseteq n$ and $\mathsf{safe}^{k}_{\mathbb{P}_1,\mathbb{I}_1}(\mathtt{st}_1', n, q)$. Since $\mathbb{P}_1 \subseteq \{p\}\cup\mathbb{P}$, we can again pick n.
(2c) We have $\mathsf{safe}^{k+1}_{\mathbb{P}_2,\mathbb{I}_2}(\mathtt{st}_2, q, o)$ by assumption. By monotonicity in Lemma 44, $\mathsf{safe}^{k}_{\mathbb{P}_2,\mathbb{I}_2}(\mathtt{st}_2, q, o)$. We already noticed $\mathsf{safe}^{k}_{\mathbb{P}_1,\mathbb{I}_1}(\mathtt{st}_1', n, q)$. The induction hypothesis yields $\mathsf{safe}^{k}_{\{q\}\cup\mathbb{P},\mathbb{I}}(\mathtt{st}_1';\mathtt{st}_2, n, o)$. □

E.2 Soundness of COM-SEM

We prove a fact that will be helpful.

Lemma 47. $\mathsf{safe}^{k}_{\{p\},\varnothing}(\mathtt{skip}, p, p)$.

Proof. Consider p. We proceed by induction on k.
$k = 0$: Done.
$k + 1$: The induction hypothesis is

$$\mathsf{safe}^{k}_{\{p\},\varnothing}(\mathtt{skip}, p, p)\ .$$

Our goal is to prove

$$\mathsf{safe}^{k+1}_{\{p\},\varnothing}(\mathtt{skip}, p, p)\ .$$

(1) We have $p \subseteq p$.
(2) The only transition is $\mathtt{skip} \xrightarrow{\mathtt{skip}} \mathtt{skip}$. As $\neg\mathsf{eff}(p, \mathtt{skip})$, we have $\mathsf{inter}(p, \mathtt{skip}) = \varnothing$.
(2b) As predicate q we pick p, which is in $\{p\}$. Then $[\![\mathtt{skip}]\!](p) = p$.
(2c) We have $\mathsf{safe}^{k}_{\mathbb{P},\mathbb{I}}(\mathtt{skip}, p, p)$ by induction. □

Soundness of COM-SEM is formalized by the following lemma.

Lemma 48. *$\mathbb{P} = \{q\}$ and $\mathbb{I} = \mathsf{inter}(p, \mathtt{com})$ and $[\![\mathtt{com}]\!](p) \subseteq q$ imply $\mathsf{safe}^{k}_{\mathbb{P},\mathbb{I}}(\mathtt{com}, p, q)$.*

Proof. We consider $k + 1$, for $k = 0$ there is nothing to do. We show

$$\mathsf{safe}^{k+1}_{\mathbb{P},\mathbb{I}}(\mathtt{com}, p, q)\ .$$

(1) If $\mathtt{com} = \mathtt{skip}$, we have $p = [\![\mathtt{com}]\!](p)$. By the assumption, $[\![\mathtt{com}]\!](p) \subseteq q$. Hence, $p \subseteq q$ as required.
(2) Consider $\mathtt{com} \xrightarrow{\mathtt{com}} \mathtt{skip}$.

(2a) We have $\mathsf{inter}(p, \mathtt{com}) = \mathbb{I}$ by assumption.
(2b) We pick $n = q$, which is in $\mathbb{P} = \{q\}$. Then $[\![\mathtt{com}]\!](p) \subseteq q$ by the assumption.
(2c) We have $\{q\} = \mathbb{P}$. Then $\mathsf{safe}^k_{\mathbb{P},\varnothing}(\mathtt{skip}, q, q)$ by Lemma 47. Monotonicity in Lemma 44 yields $\mathsf{safe}^k_{\mathbb{P},\mathbb{I}}(\mathtt{skip}, q, q)$. □

E.3 Soundness of CHOICE

LEMMA 49. *If* $\mathsf{safe}^k_{\mathbb{P}_1,\mathbb{I}_1}(\mathtt{st}_1, p, q)$ *and* $\mathsf{safe}^k_{\mathbb{P}_2,\mathbb{I}_2}(\mathtt{st}_2, p, q)$ *and* $\mathbb{P} = \mathbb{P}_1 \cup \mathbb{P}_2$ *and* $\mathbb{I} = \mathbb{I}_1 \cup \mathbb{I}_2$*, then* $\mathsf{safe}^k_{\mathbb{P},\mathbb{I}}(\mathtt{st}_1 + \mathtt{st}_2, p, q)$.

PROOF. Consider $\mathtt{st}_1$, $\mathtt{st}_2$, and q. We proceed by induction on k.
$k = 0$: Done.
$k + 1$: The induction hypothesis is

$$\forall p.\quad \mathsf{safe}^k_{\mathbb{P}_1,\mathbb{I}_1}(\mathtt{st}_1, p, q) \wedge \mathsf{safe}^k_{\mathbb{P}_2,\mathbb{I}_2}(\mathtt{st}_2, p, q) \implies \mathsf{safe}^k_{\mathbb{P},\mathbb{I}}(\mathtt{st}_1 + \mathtt{st}_2, p, q)\ .$$

Consider p with

$$\mathsf{safe}^{k+1}_{\mathbb{P}_1,\mathbb{I}_1}(\mathtt{st}_1, p, q) \wedge \mathsf{safe}^{k+1}_{\mathbb{P}_2,\mathbb{I}_2}(\mathtt{st}_2, p, q)\ .$$

Our goal is to show

$$\mathsf{safe}^{k+1}_{\mathbb{P},\mathbb{I}}(\mathtt{st}_1 + \mathtt{st}_2, p, q)\ .$$

(1) We have $\mathtt{st}_1 + \mathtt{st}_2 \neq \mathtt{skip}$, hence there is nothing to show.
(2a) We have $\mathtt{st}_1 + \mathtt{st}_2 \xrightarrow{\mathtt{skip}} \mathtt{st}_i$ with $i = 1, 2$. As $\neg\mathsf{eff}(p, \mathtt{skip})$, we have $\mathsf{inter}(p, \mathtt{com}) = \varnothing \subseteq \mathbb{I}$.
(2b) As $\mathsf{safe}^{k+1}_{\mathbb{P}_1,\mathbb{I}_i}(\mathtt{st}_i, p, q)$ by assumption, there is $n \in \mathbb{P}_i$ as follows. We have $[\![\mathtt{com}]\!](p) \subseteq n$ and $\mathsf{safe}^k_{\mathbb{P}_i,\mathbb{I}_i}(\mathtt{st}_i, n, q)$. As $\mathbb{P}_i \subseteq \mathbb{P}$, we can again pick n.
(2c) Moreover, $\mathsf{safe}^k_{\mathbb{P}_i,\mathbb{I}_i}(\mathtt{st}_i, n, q)$ implies $\mathsf{safe}^k_{\mathbb{P},\mathbb{I}}(\mathtt{st}_i, n, q)$ by monotonicity in Lemma 44. □

E.4 Soundness of INFER-SEM

Follows directly from monotonicity in Lemma 44.

E.5 Soundness of LOOP

We first prove some helpful auxiliary lemmas.

LEMMA 50. *If we have* $r \in \mathbb{P}$ *and* $\mathsf{safe}^k_{\mathbb{P},\mathbb{I}}(\mathtt{st};\mathtt{st}^*, r, p)$ *and* $\mathsf{safe}^k_{\mathbb{P},\mathbb{I}}(\mathtt{skip}, r, p)$*, then* $\mathsf{safe}^{k+1}_{\mathbb{P},\mathbb{I}}(\mathtt{st}^*, r, p)$.

PROOF. Consider $\mathtt{st}$ and p. We proceed by induction on k.
$k = 0$: Done.
$k + 1$: The induction hypothesis is:

$$\forall r \in \mathbb{P}.\quad \mathsf{safe}^k_{\mathbb{P},\mathbb{I}}(\mathtt{st};\mathtt{st}^*, r, p) \wedge \mathsf{safe}^k_{\mathbb{P},\mathbb{I}}(\mathtt{skip}, r, p) \implies \mathsf{safe}^{k+1}_{\mathbb{P},\mathbb{I}}(\mathtt{st}^*, r, p)\ .$$

Consider $r \in \mathbb{P}$ so that

$$\mathsf{safe}^{k+1}_{\mathbb{P},\mathbb{I}}(\mathtt{st};\mathtt{st}^*, r, p) \wedge \mathsf{safe}^{k+1}_{\mathbb{P},\mathbb{I}}(\mathtt{skip}, r, p)\ .$$

We show

$$\mathsf{safe}^{k+2}_{\mathbb{P},\mathbb{I}}(\mathtt{st}^*, r, p)\ .$$

(1) As $\mathtt{st}^* \neq \mathtt{skip}$, there is nothing to show.
(2) The only transition is $\mathtt{st}^* \xrightarrow{\mathtt{skip}} \mathtt{skip} + \mathtt{st};\mathtt{st}^*$.
(2a) As $\neg\mathsf{eff}(r, \mathtt{skip})$, we have $\mathsf{inter}(r, \mathtt{com}) = \varnothing$ and hence $\mathsf{inter}(r, \mathtt{com}) \subseteq \mathbb{I}$.
(2b) As predicate n we choose r, of which we know it is in $\mathbb{P}$. We have $[\![\mathtt{skip}]\!](r) = r$.

(2c) By assumption, we have $\mathsf{safe}^{k+1}_{\mathbb{P},\mathbb{I}}(\mathtt{st};\mathtt{st}^*, r, p)$ as well as $\mathsf{safe}^{k+1}_{\mathbb{P},\mathbb{I}}(\mathtt{skip}, r, p)$. With Lemma 49, we get $\mathsf{safe}^{k+1}_{\mathbb{P},\mathbb{I}}(\mathtt{skip} + \mathtt{st};\mathtt{st}^*, r, p)$.
This concludes the proof of $\mathsf{safe}^{k+2}_{\mathbb{P},\mathbb{I}}(\mathtt{st}^*, r, p)$. □

Lemma 51. *If* $\mathsf{safe}^{k}_{\mathbb{P},\mathbb{I}}(\mathtt{st}_1, r, p)$ *and* $\mathsf{safe}^{k}_{\{p\}\cup\mathbb{P},\mathbb{I}}(\mathtt{st}^*, p, p)$ *hold, then* $\mathsf{safe}^{k}_{\{p\}\cup\mathbb{P},\mathbb{I}}(\mathtt{st}_1;\mathtt{st}^*, r, p)$.

Proof. Consider $\mathtt{st}$ and p. We proceed by induction on k.
$k = 0$: There is nothing to do.
$k + 1$: The induction hypothesis is

$$\forall \mathtt{st}_1.\forall r.\quad \mathsf{safe}^{k}_{\mathbb{P},\mathbb{I}}(\mathtt{st}_1, r, p) \wedge \mathsf{safe}^{k}_{\{p\}\cup\mathbb{P},\mathbb{I}}(\mathtt{st}^*, p, p) \implies \mathsf{safe}^{k}_{\{p\}\cup\mathbb{P},\mathbb{I}}(\mathtt{st}_1;\mathtt{st}^*, r, p)\ .$$

Consider $\mathtt{st}_1$ and r so that

$$\mathsf{safe}^{k+1}_{\mathbb{P},\mathbb{I}}(\mathtt{st}_1, r, p) \wedge \mathsf{safe}^{k+1}_{\{p\}\cup\mathbb{P},\mathbb{I}}(\mathtt{st}^*, p, p).$$

We show

$$\mathsf{safe}^{k+1}_{\{p\}\cup\mathbb{P},\mathbb{I}}(\mathtt{st}_1;\mathtt{st}^*, r, p).$$

(1) As $\mathtt{st}_1;\mathtt{st}^* \neq \mathtt{skip}$, there is nothing to show.
(2) Consider $\mathtt{st}_1;\mathtt{st}^* \xrightarrow{\mathsf{com}} \mathtt{st}'$. There are two cases for transitions from $\mathtt{st}_1;\mathtt{st}^*$.

Case 1: We have $\mathtt{st}_1 = \mathtt{skip}$ and $\mathtt{st}_1;\mathtt{st}^* \xrightarrow{\mathsf{skip}} \mathtt{st}^*$.
(2a) Since $\neg\mathsf{eff}(r, \mathtt{skip})$, we have $\mathsf{inter}(r, \mathtt{skip}) = \emptyset \subseteq \mathbb{I}$.
(2b) We argue that p is the right predicate to pick as n. It is in $\{p\} \cup \mathbb{P}$. We have $[\![\mathtt{skip}]\!](r) = r$. By $\mathsf{safe}^{k+1}_{\mathbb{P},\mathbb{I}}(\mathtt{st}_1, r, p) = \mathsf{safe}^{k+1}_{\mathbb{P},\mathbb{I}}(\mathtt{skip}, r, p)$, we have $r \subseteq p$. Hence, $[\![\mathtt{skip}]\!](r) \subseteq p$.
(2c) By assumption, $\mathsf{safe}^{k+1}_{\{p\}\cup\mathbb{P},\mathbb{I}}(\mathtt{st}^*, p, p)$. By Lemma 44, we get the desired $\mathsf{safe}^{k}_{\{p\}\cup\mathbb{P},\mathbb{I}}(\mathtt{st}^*, p, p)$.

Case 2: We have $\mathtt{st}_1;\mathtt{st}^* \xrightarrow{\mathsf{com}} \mathtt{st}'$ due to $\mathtt{st}_1 \xrightarrow{\mathsf{com}} \mathtt{st}'_1$ and $\mathtt{st}' = \mathtt{st}'_1;\mathtt{st}^*$.
(2a) Since $\mathsf{safe}^{k+1}_{\mathbb{P},\mathbb{I}}(\mathtt{st}_1, r, p)$, we have $\mathsf{inter}(r, \mathsf{com}) \subseteq \mathbb{I}$.
(2b) By the same assumption, there is some $n \in \mathbb{P}$ such that $[\![\mathsf{com}]\!](r) \subseteq n$ and $\mathsf{safe}^{k}_{\mathbb{P},\mathbb{I}}(\mathtt{st}'_1, n, p)$.
(2c) Since $\mathsf{safe}^{k+1}_{\{p\}\cup\mathbb{P},\mathbb{I}}(\mathtt{st}^*, p, p)$, we have $\mathsf{safe}^{k}_{\{p\}\cup\mathbb{P},\mathbb{I}}(\mathtt{st}^*, p, p)$ with Lemma 44. We apply the induction hypothesis to $\mathsf{safe}^{k}_{\mathbb{P},\mathbb{I}}(\mathtt{st}'_1, n, p)$ and $\mathsf{safe}^{k}_{\{p\}\cup\mathbb{P},\mathbb{I}}(\mathtt{st}^*, p, p)$. It yields $\mathsf{safe}^{k}_{\{p\}\cup\mathbb{P},\mathbb{I}}(\mathtt{st}_1;\mathtt{st}^*, n, p)$.
This concludes the proof of $\mathsf{safe}^{k+1}_{\{p\}\cup\mathbb{P},\mathbb{I}}(\mathtt{st}_1;\mathtt{st}^*, r, p)$. □

Soundness of Loop is the following.

Lemma 52. *If* $\mathsf{safe}^{k}_{\mathbb{P},\mathbb{I}}(\mathtt{st}, p, p)$, *then* $\mathsf{safe}^{k}_{\{p\}\cup\mathbb{P},\mathbb{I}}(\mathtt{st}^*, p, p)$.

Proof. Consider $\mathbb{P}$, $\mathbb{I}$, p, and $\mathtt{st}$. We proceed by induction on k.
$k = 0$: Done.
$k + 1$: The induction hypothesis is

$$\mathsf{safe}^{k}_{\mathbb{P},\mathbb{I}}(\mathtt{st}, p, p) \quad\Rightarrow\quad \mathsf{safe}^{k}_{\{p\}\cup\mathbb{P},\mathbb{I}}(\mathtt{st}^*, p, p).$$

We assume

$$\mathsf{safe}^{k+1}_{\mathbb{P},\mathbb{I}}(\mathtt{st}, p, p)$$

Our goal is to show

$$\mathsf{safe}^{k+1}_{\{p\}\cup\mathbb{P},\mathbb{I}}(\mathtt{st}^*, p, p).$$

By assumption, we have $\mathsf{safe}^{k+1}_{\mathbb{P},\mathbb{I}}(\mathtt{st}, p, p)$. This yields $\mathsf{safe}^{k}_{\mathbb{P},\mathbb{I}}(\mathtt{st}, p, p)$ by Lemma 44. The induction hypothesis yields $\mathsf{safe}^{k}_{\{p\}\cup\mathbb{P},\mathbb{I}}(\mathtt{st}^*, p, p)$. Lemma 51 yields $\mathsf{safe}^{k}_{\{p\}\cup\mathbb{P},\mathbb{I}}(\mathtt{st};\mathtt{st}^*, p, p)$. Lemma 47 yields $\mathsf{safe}^{k}_{\{p\},\varnothing}(\mathtt{skip}, p, p)$. Monotonicity in Lemma 44 yields $\mathsf{safe}^{k}_{\{p\}\cup\mathbb{P},\mathbb{I}}(\mathtt{skip}, p, p)$. We can thus invoke Lemma 50. It yields the required $\mathsf{safe}^{k+1}_{\{p\}\cup\mathbb{P},\mathbb{I}}(\mathtt{st}^*, p, p)$. □

E.6 Soundness of FRAME

We first prove an auxiliary lemma stating that due to the locality of commands, effectfullness of commands is compatible with framing.

Lemma 53. $\mathsf{eff}(o * p, \mathsf{com})$ *implies* $\mathsf{eff}(o, \mathsf{com})$.

Proof. We prove the contrapositive:

$$\neg\mathsf{eff}(o, \mathsf{com}) \implies \neg\mathsf{eff}(o * p, \mathsf{com}) .$$

Consider some $(\mathsf{g}_1, \mathsf{l}_1) \in o$ and some $(\mathsf{g}_2, \mathsf{l}_2) \in p$ such that $(\mathsf{g}_1 * \mathsf{g}_2, \mathsf{l}_1 * \mathsf{l}_2) = (\mathsf{g}_1, \mathsf{l}_1) * (\mathsf{g}_2, \mathsf{l}_2) \in o * p$. By the locality of commands, we have

$$[\![\mathsf{com}]\!]((\mathsf{g}_1, \mathsf{l}_1) * (\mathsf{g}_2, \mathsf{l}_2)) \subseteq [\![\mathsf{com}]\!](\mathsf{g}_1, \mathsf{l}_1) * \{(\mathsf{g}_2, \mathsf{l}_2)\} .$$

Since $\neg\mathsf{eff}(o, \mathsf{com})$ and $(\mathsf{g}_1, \mathsf{l}_1) \in o$, every state in $[\![\mathsf{com}]\!](\mathsf{g}_1, \mathsf{l}_1)$ takes the form $(\mathsf{g}_1, \mathsf{l})$ for some l. Consequently, every state in $[\![\mathsf{com}]\!]((\mathsf{g}_1, \mathsf{l}_1) * (\mathsf{g}_2, \mathsf{l}_2))$ must be of the following form: $(\mathsf{g}_1, \mathsf{l}) * (\mathsf{g}_2, \mathsf{l}_2) = (\mathsf{g}_1 * \mathsf{g}_2, \mathsf{l} * \mathsf{l}_2)$, and is thus as required. □

Soundness of FRAME is this.

Lemma 54. *If* $\mathsf{safe}^{k}_{\mathbb{P},\mathbb{I}}(\mathtt{st}, r, q)$, *then* $\mathsf{safe}^{k}_{\mathbb{P} * o,\mathbb{I} * o}(\mathtt{st}, r * o, q * o)$.

Proof. Consider q, $\mathbb{I}$, $\mathbb{P}$, and o. We proceed by induction on k.
$k = 0$: There is nothing to do.
$k + 1$: The induction hypothesis is

$$\forall \mathtt{st}. \forall r.\ \ \mathsf{safe}^{k}_{\mathbb{P},\mathbb{I}}(\mathtt{st}, r, q) \ \Rightarrow\ \mathsf{safe}^{k}_{\mathbb{P} * o,\mathbb{I} * o}(\mathtt{st}, r * o, q * o).$$

Consider $\mathtt{st}$ and r with $\mathsf{safe}^{k+1}_{\mathbb{P},\mathbb{I}}(\mathtt{st}, r, q)$. We show

$$\mathsf{safe}^{k+1}_{\mathbb{P},\mathbb{I}}(\mathtt{st}, r * o, q * o).$$

(1) Assume $\mathtt{st} = \mathtt{skip}$, otherwise there is nothing to do. Since $\mathsf{safe}^{k+1}_{\mathbb{P},\mathbb{I}}(\mathtt{st}, r, q)$, we have $r \subseteq q$. Hence, $r * o \subseteq q * o$, as required.
(2) Consider com and $\mathtt{st}'$ so that $\mathtt{st} \xrightarrow{\mathsf{com}} \mathtt{st}'$. By assumption, we have $\mathsf{safe}^{k+1}_{\mathbb{P},\mathbb{I}}(\mathtt{st}, r, q)$.
(2a) If $\mathsf{inter}(r * o, \mathsf{com}) = \varnothing$, then $\mathsf{inter}(r * o, \mathsf{com}) \subseteq \mathbb{I} * o$ follows trivially. So assume that we have $\mathsf{inter}(r * o, \mathsf{com}) = \{(r * o, \mathsf{com})\}$. Then $\mathsf{eff}(r * o, \mathsf{com})$. By Lemma 53, we get $\mathsf{eff}(r, \mathsf{com})$. Hence, $\mathsf{inter}(r, \mathsf{com}) = \{(r, \mathsf{com})\}$. Since $\mathsf{safe}^{k+1}_{\mathbb{P},\mathbb{I}}(\mathtt{st}, r, q)$, we have $\mathsf{inter}(r, \mathsf{com}) \subseteq \mathbb{I}$. This means there is $(o', \mathsf{com}) \in \mathbb{I}$ with $r \subseteq o'$. Then $r * o \subseteq o' * o$. Moreover, $(o' * o, \mathsf{com}) \in \mathbb{I} * o$, as required.
(2b) Because of $\mathsf{safe}^{k+1}_{\mathbb{P},\mathbb{I}}(\mathtt{st}, r, q)$, there is $n \in \mathbb{P}$ with $[\![\mathsf{com}]\!](r) \subseteq n$ and $\mathsf{safe}^{k}_{\mathbb{P},\mathbb{I}}(\mathtt{st}', n, q)$. We have $n * o \in \mathbb{P} * o$. We argue that this is the right choice for a predicate. By $[\![\mathsf{com}]\!](r) \subseteq n$ and the locality of commands, $[\![\mathsf{com}]\!](r * o) \subseteq n * o$, as required.
(2c) We have $\mathsf{safe}^{k}_{\mathbb{P},\mathbb{I}}(\mathtt{st}', n, q)$. Induction yields the desired $\mathsf{safe}^{k}_{\mathbb{P},\mathbb{I}}(\mathtt{st}', n * o, q * o)$. □

E.7 Interference-Freedom

Interference freedom lifts the safety guarantee from an isolated thread to the concurrency library.

Definition 55. *We define* $\mathsf{cfsafe}^0(\mathsf{cf}, q) \triangleq$ *true as well as* $\mathsf{cfsafe}^{\mathsf{k}+1}(\mathsf{cf}, q) \triangleq (1) \wedge (2)$ *with*

(1) $\mathsf{cf} \in \mathsf{Acc}_q$

(2) $\forall \mathsf{cf}'.\ \mathsf{cf} \rightarrow \mathsf{cf}' \ \Rightarrow\ \mathsf{cfsafe}^{\mathsf{k}}(\mathsf{cf}', q)$.

The key lemma for lifting the thread-local safety guarantee to configurations is the following.

Lemma 56. *If* $\mathsf{cf} = (\mathsf{g}, \mathsf{pc})$ *and* $[\forall i. \forall \mathsf{l}. \forall \mathsf{st}. \mathsf{pc}(i) = (\mathsf{l}, \mathsf{st}) \Rightarrow \exists r \in \mathbb{P}.\ (\mathsf{g}, \mathsf{l}) \in r \wedge \mathsf{safe}^k_{\mathbb{P},\mathbb{I}}(\mathsf{st}, r, q)]$, *then* $\mathsf{cfsafe}^{\mathsf{k}}(\mathsf{cf}, q)$.

Proof. Consider $\mathbb{P}$ and $\mathbb{I}$ with $\boxtimes_{\mathbb{I}} \mathbb{P}$. Consider q. We proceed by induction on k.
$k = 0$: Done.
$k + 1$: The induction hypothesis is

$$\forall \mathsf{g}. \forall \mathsf{pc}.\quad \Big(\forall i. \forall \mathsf{l}. \forall \mathsf{st}. \mathsf{pc}(i) = (\mathsf{l}, \mathsf{st}) \ \Rightarrow\ \exists r \in \mathbb{P}.\ (\mathsf{g}, \mathsf{l}) \in r \wedge \mathsf{safe}^k_{\mathbb{P},\mathbb{I}}(\mathsf{st}, r, q)\Big)$$
$$\Rightarrow \mathsf{cfsafe}^{\mathsf{k}}((\mathsf{g}, \mathsf{pc}), q).$$

Consider $\mathsf{cf} = (\mathsf{g}, \mathsf{pc})$ so that for all $i, \mathsf{l}, \mathsf{st}$ we have

$$\mathsf{pc}(i) = (\mathsf{l}, \mathsf{st}) \Rightarrow \exists r \in \mathbb{P}.\ (\mathsf{g}, \mathsf{l}) \in r \wedge \mathsf{safe}^{k+1}_{\mathbb{P},\mathbb{I}}(\mathsf{st}, r, q).$$

We show $\mathsf{cfsafe}^{\mathsf{k}+1}(\mathsf{cf}, q)$.
(1) To show $\mathsf{cf} \in \mathsf{Acc}_q$, let thread i be with $\mathsf{pc}(i) = (\mathsf{l}, \mathsf{skip})$. By assumption, there is a predicate $r \in \mathbb{P}$ so that $(\mathsf{g}, \mathsf{l}) \in r$ and $\mathsf{safe}^{k+1}_{\mathbb{P},\mathbb{I}}(\mathsf{skip}, r, q)$. By definition of $\mathsf{safe}^{k+1}_{\mathbb{P},\mathbb{I}}(\mathsf{skip}, r, q)$, we have $r \subseteq q$. Hence, $(\mathsf{g}, \mathsf{l}) \in q$, as required.
(2) Consider a configuration cf' with $\mathsf{cf} \rightarrow \mathsf{cf}' = (\mathsf{g}', \mathsf{pc}')$. To establish $\mathsf{cfsafe}^{\mathsf{k}}(\mathsf{cf}', q)$, let thread i be with $\mathsf{pc}(i) = (\mathsf{l}, \mathsf{st})$ and $\mathsf{pc}'(i) = (\mathsf{l}', \mathsf{st}')$. We show that there is a predicate $n \in \mathbb{P}$ with $(\mathsf{g}', \mathsf{l}') \in n$ and $\mathsf{safe}^k_{\mathbb{P},\mathbb{I}}(\mathsf{st}', n, q)$. The induction hypothesis then yields $\mathsf{cfsafe}^{\mathsf{k}}(\mathsf{cf}', q)$ and concludes the proof. There are two cases.
Case 1: Thread i executes the command $\mathsf{st} \xrightarrow{\mathsf{com}} \mathsf{st}'$ that leads to the transition $\mathsf{cf} \rightarrow \mathsf{cf}'$. Then $(\mathsf{g}', \mathsf{l}') \in [\![\mathsf{com}]\!](\mathsf{g}, \mathsf{l})$. By assumption, there is a predicate $r \in \mathbb{P}$ so that $(\mathsf{g}, \mathsf{l}) \in r$ and $\mathsf{safe}^{k+1}_{\mathbb{P},\mathbb{I}}(\mathsf{st}, r, q)$. The definition of safety gives some $n \in \mathbb{P}$ with $[\![\mathsf{com}]\!](r) \subseteq n$. Then $(\mathsf{g}', \mathsf{l}') \in [\![\mathsf{com}]\!](\mathsf{g}, \mathsf{l})$, $(\mathsf{g}, \mathsf{l}) \in r$, and $[\![\mathsf{com}]\!](r) \subseteq n$ together entail $(\mathsf{g}', \mathsf{l}') \in n$. Moreover, the predicate n satisfies $\mathsf{safe}^k_{\mathbb{P},\mathbb{I}}(\mathsf{st}', n, q)$, as required. In the second case, we use that safety of the executing thread also yields $\mathsf{inter}(r, \mathsf{com}) \subseteq \mathbb{I}$. Hence, if $\mathsf{eff}(r, \mathsf{com})$, then there is $(o, \mathsf{com}) \in \mathbb{I}$ with $r \subseteq o$.
Case 2: Thread i experiences the command as an interference. Hence, $\mathsf{pc}'(i) = \mathsf{pc}(i) = (\mathsf{l}, \mathsf{st})$ but potentially $\mathsf{g}' \neq \mathsf{g}$. By assumption, there is a predicate $r \in \mathbb{P}$ so that $(\mathsf{g}, \mathsf{l}) \in r$ and $\mathsf{safe}^{k+1}_{\mathbb{P},\mathbb{I}}(\mathsf{st}, r, q)$. We show that $n = r$ is the right choice. We already argued that the interference is covered by some $(o, \mathsf{com}) \in \mathbb{I}$. Hence, we have $(\mathsf{g}', \mathsf{l}) \in [\![(o, \mathsf{com})]\!](\mathsf{g}, \mathsf{l})$. Moreover $[\![(o, \mathsf{com})]\!](\mathsf{g}, \mathsf{l}) \subseteq [\![(o, \mathsf{com})]\!](r)$. By interference freedom, we have $[\![(o, \mathsf{com})]\!](r) \subseteq r$. Hence, $(\mathsf{g}', \mathsf{l}) \in r$. Moreover, $\mathsf{safe}^{k+1}_{\mathbb{P},\mathbb{I}}(\mathsf{st}, r, q)$ entails the desired $\mathsf{safe}^k_{\mathbb{P},\mathbb{I}}(\mathsf{st}, r, q)$ by Lemma 44. □

E.8 Soundness

Finally, we can prove the overall soundness theorem.

Proof of Theorem 3. Assume $\mathbb{P}, \mathbb{I} \Vdash \{\, p \,\}\ \mathsf{st}\ \{\, q \,\}$ and $\boxtimes_{\mathbb{I}} \mathbb{P}$ and $p \in \mathbb{P}$. In order to show that $\models \{\, p \,\}\ \mathsf{st}\ \{\, q \,\}$ holds, consider a configuration $(\mathsf{g}, \mathsf{pc}) \in \mathsf{Init}_{p,\mathsf{st}}$. By definition of $\mathsf{Init}_{p,\mathsf{st}}$, every thread i satisfies $\mathsf{pc}(i) = (\mathsf{l}, \mathsf{st})$ with $(\mathsf{g}, \mathsf{l}) \in p$. By $\mathbb{P}, \mathbb{I} \Vdash \{\, p \,\}\ \mathsf{st}\ \{\, q \,\}$ and Proposition 45, we have

$\mathrm{safe}^k_{\mathbb{P},\mathbb{I}}(\mathrm{st}, p, q)$ for every k. With $\boxplus_{\mathbb{I}}\,\mathbb{P}$, $p \in \mathbb{P}$, and Lemma 56, we get $\mathrm{cfsafe}^k((\mathrm{g}, \mathrm{pc}), q)$ for all k. This shows that every reachable configuration is accepting for q. □

F PROOFS OF SECTION 6

We will first prove an auxiliary lemma, which we will use to show the soundness of F-INFER.

Lemma 57. *$p_2 \subseteq p_1$ and $q_1 \subseteq q_2$ imply $p_1 \mathrel{-\!\!\circledast} q_1 \subseteq p_2 \mathrel{-\!\!\circledast} q_2$.*

Proof. Consider $\mathrm{s} \in p_1 \mathrel{-\!\!\circledast} q_1$. Then $\{\mathrm{s}\} * p_1 \subseteq q_1$. As $p_2 \subseteq p_1$, we get $\{\mathrm{s}\} * p_2 \subseteq q_1$. As $q_1 \subseteq q_2$, we get $\{\mathrm{s}\} * p_2 \subseteq q_2$. This means $\mathrm{s} \in p_2 \mathrel{-\!\!\circledast} q_2$. □

Proof of Lemma 6. We prove validity of each of the rules in succession.

(i) *Soundness of* F-INTRO. We have $\mathrm{emp} * p = p$. Moreover, we have:

$$\langle\, p \,\rangle\, \mathtt{com} \,\langle\, q \,\rangle = p \mathrel{-\!\!\circledast} wp(\mathtt{com}, q) = \{\, \mathrm{s} \;\mid\; \{\mathrm{s}\} * p \subseteq wp(\mathtt{com}, q) \,\}$$

We thus have to show $\mathrm{emp} * p = p \subseteq wp(\mathtt{com}, q)$. This holds by assumption.

(ii) *Soundness of* F-SEQ. Consider $\mathrm{s} \in \langle\, p \,\rangle\, \mathtt{com}_1 \,\langle\, q \,\rangle * \langle\, q \,\rangle\, \mathtt{com}_2 \,\langle\, o \,\rangle$. Then we have $\mathrm{s} = \mathrm{s}_1 * \mathrm{s}_2$ with $\mathrm{s}_1 \in \langle\, p \,\rangle\, \mathtt{com}_1 \,\langle\, q \,\rangle$ and $\mathrm{s}_2 \in \langle\, q \,\rangle\, \mathtt{com}_2 \,\langle\, o \,\rangle$. So we arrive at $\{\mathrm{s}\} * p = \{\mathrm{s}_2\} * (\{\mathrm{s}_1\} * p)$. Then, by the locality of commands, we obtain $[\![\mathtt{com}_1]\!](\{\mathrm{s}\} * p) \subseteq \{\mathrm{s}_2\} * [\![\mathtt{com}_1]\!](\{\mathrm{s}_1\} * p)$. As $\mathrm{s}_1 \in \langle\, p \,\rangle\, \mathtt{com}_1 \,\langle\, q \,\rangle$, we have $[\![\mathtt{com}_1]\!](\{\mathrm{s}_1\} * p) \subseteq q$. Hence, we get $[\![\mathtt{com}_1]\!](\{\mathrm{s}\} * p) \subseteq \{\mathrm{s}_2\} * q$. This means we arrive at $\{\mathrm{s}\} * p \subseteq wp(\mathtt{com}_1, \{\mathrm{s}_2\} * q)$. As $\mathrm{s}_2 \in \langle\, q \,\rangle\, \mathtt{com}_2 \,\langle\, o \,\rangle$, we have $\{\mathrm{s}_2\} * q \subseteq wp(\mathtt{com}_2, o)$. Together:

$$\{\mathrm{s}\} * p \subseteq wp(\mathtt{com}_1, \{\mathrm{s}_2\} * q) \subseteq wp(\mathtt{com}_1, wp(\mathtt{com}_2, o)) = wp(\mathtt{com}_1; \mathtt{com}_2, o)$$

By assumption, $wp(\mathtt{com}_1; \mathtt{com}_2, o) \subseteq wp(\mathtt{com}, o)$. Hence, we arrive at $\{\mathrm{s}\} * p \subseteq wp(\mathtt{com}, o)$. This shows $\mathrm{s} \in \langle\, p \,\rangle\, \mathtt{com} \,\langle\, o \,\rangle$, as required.

(iii) *Soundness of* F-ACCOUNT. Consider $\mathrm{s} \in p * \langle\, p * q \,\rangle\, \mathtt{com} \,\langle\, o \,\rangle$. Then $\mathrm{s} = \mathrm{s}_1 * \mathrm{s}_2$ with $\mathrm{s}_1 \in p$ and $\mathrm{s}_2 \in \langle\, p * q \,\rangle\, \mathtt{com} \,\langle\, o \,\rangle$. The latter means $[\![\mathtt{com}]\!](\{\mathrm{s}_2\} * p * q) \subseteq o$. As $\mathrm{s}_1 \in p$, we in particular have $[\![\mathtt{com}]\!](\{\mathrm{s}_2\} * \{\mathrm{s}_1\} * q) \subseteq o$. Since $[\![\mathtt{com}]\!](\{\mathrm{s}_2\} * \{\mathrm{s}_1\} * q) = [\![\mathtt{com}]\!](\{\mathrm{s}\} * q)$, we obtain the required $\mathrm{s} \in \langle\, q \,\rangle\, \mathtt{com} \,\langle\, o \,\rangle$.

(iv) *Soundness of* F-INFER. If $q_1 \subseteq q_2$, then $wp(\mathtt{com}, q_1) \subseteq wp(\mathtt{com}, q_2)$. We conclude as follows:

$$\begin{array}{rcl} & & \langle\, p_1 \,\rangle\, \mathtt{com} \,\langle\, q_1 \,\rangle \\ (\text{ Definition 5 }) & = & p_1 \mathrel{-\!\!\circledast} wp(\mathtt{com}, q_1) \\ (\text{ Lemma 57 and assumption }) & \subseteq & p_2 \mathrel{-\!\!\circledast} wp(\mathtt{com}, q_1) \\ (\text{ Lemma 57 and remark above }) & \subseteq & p_2 \mathrel{-\!\!\circledast} wp(\mathtt{com}, q_2) \\ (\text{ Definition 5 }) & = & \langle\, p_2 \,\rangle\, \mathtt{com} \,\langle\, q_2 \,\rangle\, . \end{array}$$

(v) *Soundness of* F-FRAME. Consider $\mathrm{s} \in \langle\, p \,\rangle\, \mathtt{com} \,\langle\, q \,\rangle$. Then $\{\mathrm{s}\} * p \subseteq wp(\mathtt{com}, q)$. This in turn means $[\![\mathtt{com}]\!](\{\mathrm{s}\} * p) \subseteq q$. By the locality of commands, we have $[\![\mathtt{com}]\!](\{\mathrm{s}\} * p * o) \subseteq q * o$. This means $\{\mathrm{s}\} * p * o \subseteq wp(\mathtt{com}, q * o)$. Hence, $\mathrm{s} \in \langle\, p * o \,\rangle\, \mathtt{com} \,\langle\, q * o \,\rangle$.

(vi) *Soundness of* F-INVOKE. We conclude by:

$$\begin{array}{rcl} & & p * \langle\, p \,\rangle\, \mathtt{com} \,\langle\, q \,\rangle \\ (\text{ emp neutral }) & = & p * \langle\, p * \mathrm{emp} \,\rangle\, \mathtt{com} \,\langle\, q \,\rangle \\ (\text{ Soundness of F-ACCOUNT }) & \subseteq & \langle\, \mathrm{emp} \,\rangle\, \mathtt{com} \,\langle\, q \,\rangle \\ (\text{ Definition 5 }) & = & \mathrm{emp} \mathrel{-\!\!\circledast} wp(\mathtt{com}, q) \\ (\text{ emp neutral }) & = & wp(\mathtt{com}, q). \end{array}$$

□

G PROOFS OF SECTION 7

Proof of Lemma 10. Consider $\sigma' \in [\![\![\mathsf{com}]\!]\!](a * c)$. Then there is $\sigma \in a * c$ with $\sigma' \in [\![\![\mathsf{com}]\!]\!](\sigma)$. Then $\sigma = \sigma_1.\mathsf{s}_1 * \sigma_2.\mathsf{s}_2$ with $\sigma_1.\mathsf{s}_1 \in a$ and $\sigma_2.\mathsf{s}_2 \in c$. By definition, $\sigma' = \sigma.\mathsf{s}'$ with $\mathsf{s}' \in [\![\mathsf{com}]\!](\mathsf{s}_1 * \mathsf{s}_2)$. By locality of commands, $[\![\mathsf{com}]\!](\mathsf{s}_1 * \mathsf{s}_2) \subseteq [\![\mathsf{com}]\!](\mathsf{s}_1) * \{\mathsf{s}_2\}$. Hence, $\mathsf{s}' = \mathsf{s}_1' * \mathsf{s}_2$ with $\mathsf{s}_1' \in [\![\mathsf{com}]\!](\mathsf{s}_1)$. By definition, $\sigma_1.\mathsf{s}_1.\mathsf{s}_1' \in [\![\![\mathsf{com}]\!]\!](\sigma_1.\mathsf{s}_1)$. Since $\sigma_1.\mathsf{s}_1 \in a$, we have $\sigma_1.\mathsf{s}_1.\mathsf{s}_1' \in [\![\![\mathsf{com}]\!]\!](a)$. By frameability, we have $\sigma_2.\mathsf{s}_2.\mathsf{s}_2 \in c$. Hence, $\sigma_1.\mathsf{s}_1.\mathsf{s}_1' * \sigma_2.\mathsf{s}_2.\mathsf{s}_2 \in [\![\![\mathsf{com}]\!]\!](a) * c$. Altogether, we arrive at $\sigma_1.\mathsf{s}_1.\mathsf{s}_1' * \sigma_2.\mathsf{s}_2.\mathsf{s}_2 = \sigma.(\mathsf{s}_1' * \mathsf{s}_2) = \sigma'$. □

We adapt the semantics of concurrency libraries to track history and show that the thread-modular reasoning principle remains sound. The first step is to lift the product separation algebra to a history separation algebra containing sequences from $(\Sigma_\mathsf{G} \times \Sigma_\mathsf{L})^+$. In a configuration of the concurrency library, we store the two components of such a sequence separately. So we store $(\mathsf{g}_1, \mathsf{l}_1) \dots (\mathsf{g}_n, \mathsf{l}_n)$ as the pair (γ, λ) with $\gamma = \mathsf{g}_1 \dots \mathsf{g}_n$ and $\lambda = \mathsf{l}_1 \dots \mathsf{l}_n$. Note that a predicate $a \subseteq (\Sigma_\mathsf{G} \times \Sigma_\mathsf{L})^+$ only contains pairs of sequences of the same length. A configuration of the concurrency library has the shape (γ, pc) with $\gamma \in \Sigma_\mathsf{G}^+$ a sequence of global states and for all threads $\mathsf{pc}(i) = (\lambda, \mathtt{st})$ with $\lambda \in \Sigma_\mathsf{L}^+$ an equally long sequence of local states. When we execute a transition, we not only update the local state of the thread being active. We also store a copy of the current local state in all other threads. This ensures the resulting sequences of global and local states are again of the same length. The modified transition relation is this.

$$\frac{\begin{array}{c}\mathsf{pc}_1(i) = (\lambda_1, \mathtt{st}_1) \qquad \mathtt{st}_1 \xrightarrow{\mathsf{com}} \mathtt{st}_2 \qquad (\gamma_2, \lambda_2) \in [\![\![\mathsf{com}]\!]\!](\gamma_1, \lambda_1) \qquad \mathsf{pc}_2(i) = (\lambda_2, \mathtt{st}_2) \\ \forall j \neq i.\ \ \mathsf{pc}_1(j) = (\lambda_j, \mathtt{st}_j) \qquad \mathsf{pc}_2(j) = (\lambda_j.\mathsf{last}(\lambda_j), \mathtt{st}_j)\end{array}}{(\gamma_1, \mathsf{pc}_1) \rightarrow (\gamma_2, \mathsf{pc}_2)}$$

We also adapt interference to copy the last local state:

$$[\![\![(o, \mathsf{com})]\!]\!](\gamma, \lambda) \triangleq \{\, (\gamma', \lambda.\mathsf{last}(\lambda)) \ \mid\ \exists \lambda_1, \lambda_2.\ \ (\gamma, \lambda_1) \in o \ \wedge\ (\gamma', \lambda_2) \in [\![\![\mathsf{com}]\!]\!](\gamma, \lambda_1) \,\}\,.$$

Proposition 45 refers to a general separation algebra and therefore continues to hold in the present setting modulo the restriction of the FRAME rule to frameable predicates.

We now show that the key lemma for lifting the thread-local safety guarantee to configurations carries over to history separation algebras.

Lemma 58. *If* $\mathsf{cf} = (\gamma, \mathsf{pc})$ *and* $[\forall i.\forall\lambda.\forall \mathtt{st}.\mathsf{pc}(i) = (\lambda, \mathtt{st}).\Rightarrow \exists d \in \mathbb{P}.\ (\gamma, \lambda) \in d \wedge \mathsf{safe}^k_{\mathbb{P},\mathbb{I}}(\mathtt{st}, d, b)]$ *and* $\boxplus_\mathbb{I}\,\mathbb{P}$ *hold, then we have* $\mathsf{cfsafe}^\mathsf{k}(\mathsf{cf}, b)$.

Proof of Lemma 58. Consider $\mathbb{P}$ and $\mathbb{I}$ with $\boxplus_\mathbb{I}\,\mathbb{P}$. Consider b. We proceed by induction on k.
$k = 0$: Done.
$k + 1$: The induction hypothesis is

$$\forall\gamma.\forall\mathsf{pc}.\ \ \Big(\forall i.\forall\lambda.\forall\mathtt{st}.\ \ \mathsf{pc}(i) = (\lambda, \mathtt{st}) \ \Rightarrow\ \exists d \in \mathbb{P}.\ (\gamma, \lambda) \in d \wedge \mathsf{safe}^k_{\mathbb{P},\mathbb{I}}(\mathtt{st}, d, b)\Big)$$
$$\Rightarrow \mathsf{cfsafe}^\mathsf{k}((\gamma, \mathsf{pc}), b).$$

Consider $\mathsf{cf} = (\gamma, \mathsf{pc})$ so that for all $i, \lambda, \mathtt{st}$ we have

$$\mathsf{pc}(i) = (\lambda, \mathtt{st}) \Rightarrow \exists d \in \mathbb{P}.\ (\gamma, \lambda) \in d \wedge \mathsf{safe}^{k+1}_{\mathbb{P},\mathbb{I}}(\mathtt{st}, d, b).$$

We show $\mathsf{cfsafe}^{\mathsf{k}+1}(\mathsf{cf}, b)$.
(1) To show $\mathsf{cf} \in \mathsf{Acc}_b$, let thread i be with $\mathsf{pc}(i) = (\lambda, \mathtt{skip})$. By assumption, there is a predicate $d \in \mathbb{P}$ so that $(\gamma, \lambda) \in d$ and $\mathsf{safe}^{k+1}_{\mathbb{P},\mathbb{I}}(\mathtt{skip}, d, b)$. By definition of $\mathsf{safe}^{k+1}_{\mathbb{P},\mathbb{I}}(\mathtt{skip}, d, b)$, we have $d \subseteq b$. Hence, $(\gamma, \lambda) \in b$, as required.
(2) Consider a configuration cf' with $\mathsf{cf} \rightarrow \mathsf{cf}' = (\gamma', \mathsf{pc}')$. We establish $\mathsf{cfsafe}^\mathsf{k}(\mathsf{cf}', b)$. Consider a thread i with $\mathsf{pc}(i) = (\lambda, \mathtt{st})$ and $\mathsf{pc}'(i) = (\lambda', \mathtt{st}')$. We show that there is a predicate $e \in \mathbb{P}$ with

$(\gamma', \lambda') \in e$ and $\mathsf{safe}^k_{\mathbb{P},\mathbb{I}}(\mathtt{st}', e, b)$. The induction hypothesis then yields $\mathsf{cfsafe}^{\mathsf{k}}(\mathsf{cf}', b)$ and concludes the proof. There are two cases.
Case 1: Thread i executes the command $\mathtt{st} \xrightarrow{\mathtt{com}} \mathtt{st}'$ that leads to the transition $\mathsf{cf} \to \mathsf{cf}'$. Then $(\gamma', \lambda') \in [\![\mathtt{com}]\!](\gamma, \lambda)$. By assumption, there is a predicate $d \in \mathbb{P}$ with $(\gamma, \lambda) \in d$ and $\mathsf{safe}^{k+1}_{\mathbb{P},\mathbb{I}}(\mathtt{st}, d, b)$. The definition of safety gives a predicate $e \in \mathbb{P}$ with $[\![\mathtt{com}]\!](d) \subseteq e$. Then $(\gamma', \lambda') \in [\![\mathtt{com}]\!](\gamma, \lambda)$, $(\gamma, \lambda) \in d$, and $[\![\mathtt{com}]\!](d) \subseteq e$ together entail $(\gamma', \lambda') \in e$. Moreover, the predicate e satisfies $\mathsf{safe}^k_{\mathbb{P},\mathbb{I}}(\mathtt{st}', e, b)$, as required. In the second case, we will use that safety of the executing thread also yields $\mathsf{inter}(d, \mathtt{com}) \subseteq \mathbb{I}$. Hence, if $\mathsf{eff}(d, \mathtt{com})$, then there is $(c, \mathtt{com}) \in \mathbb{I}$ with $d \subseteq c$.
Case 2: Thread i experiences the command as an interference. Then we have $\mathsf{pc}(i) = (\lambda, \mathtt{st})$ and $\mathsf{pc}'(i) = (\lambda.\mathsf{last}(\lambda), \mathtt{st})$. By assumption, there is a predicate $d \in \mathbb{P}$ so that $(\gamma, \lambda) \in d$ and $\mathsf{safe}^{k+1}_{\mathbb{P},\mathbb{I}}(\mathtt{st}, d, b)$. We show that $e = d$ is the right choice. We already argued that the interference is covered by some $(c, \mathtt{com}) \in \mathbb{I}$. Hence, we have $(\gamma', \lambda.\mathsf{last}(\lambda)) \in [\![(c, \mathtt{com})]\!](\gamma, \lambda)$. Moreover $[\![(c, \mathtt{com})]\!](\gamma, \lambda) \subseteq [\![(c, \mathtt{com})]\!](d)$. By interference freedom, we have $[\![(c, \mathtt{com})]\!](d) \subseteq d$. Hence, $(\gamma', \lambda.\mathsf{last}(\lambda)) \in d$. Moreover, $\mathsf{safe}^{k+1}_{\mathbb{P},\mathbb{I}}(\mathtt{st}, d, b)$ entails $\mathsf{safe}^k_{\mathbb{P},\mathbb{I}}(\mathtt{st}, d, b)$ by Lemma 44. □

The proof of Theorem 11 is the same as the one for Theorem 3, except that we replace Lemma 56 by the above Lemma 58.

Proof of Theorem 11. Assume that $\mathbb{P}, \mathbb{I} \Vdash \{\, a \,\}\ \mathtt{st}\ \{\, b \,\}$ and $\boxast_{\mathbb{I}}\, \mathbb{P}$ and $a \in \mathbb{P}$ hold. To show $\models \{\, a \,\}\ \mathtt{st}\ \{\, b \,\}$, consider a configuration $(\gamma, \mathsf{pc}) \in \mathsf{Init}_{a,\mathtt{st}}$. By definition of $\mathsf{Init}_{a,\mathtt{st}}$, every thread i satisfies $\mathsf{pc}(i) = (\lambda, \mathtt{st})$ with $(\gamma, \lambda) \in a$. By $\mathbb{P}, \mathbb{I} \Vdash \{\, a \,\}\ \mathtt{st}\ \{\, b \,\}$ together with Proposition 45, we have $\mathsf{safe}^k_{\mathbb{P},\mathbb{I}}(\mathtt{st}, a, b)$ for every k. With $\boxast_{\mathbb{I}}\, \mathbb{P}$, $a \in \mathbb{P}$, and Lemma 58, we get $\mathsf{cfsafe}^{\mathsf{k}}((\gamma, \mathsf{pc}), b)$ for all k. This shows that every reachable configuration is accepting for b. □

Proof of Lemma 13. (1) Let $\sigma.\mathsf{s} \in \_p$, which means $\mathsf{s} \in p$. Then also for $\sigma.\mathsf{s}.\mathsf{s}$ the last state belongs to p. Thus, $\sigma.\mathsf{s}.\mathsf{s} \in \_p$.
Consider $\sigma.\mathsf{s} \in \Diamond p$. This means there is a state from p in $\sigma.\mathsf{s}$. That state is still present in $\sigma.\mathsf{s}.\mathsf{s}$. Hence $\sigma.\mathsf{s}.\mathsf{s} \in \Diamond p$, as required.
(2) Consider $\sigma.\mathsf{s} \in a * b$. This means $\sigma.\mathsf{s} = \sigma_1.\mathsf{s}_1 * \sigma_2.\mathsf{s}_2$ with $\sigma_1.\mathsf{s}_1 \in a$ and $\sigma_2.\mathsf{s}_2 \in b$. As a and b are frameable, we have $\sigma_1.\mathsf{s}_1.\mathsf{s}_1 \in a$ and $\sigma_2.\mathsf{s}_2.\mathsf{s}_2 \in b$. Hence, we obtain

$$\sigma.\mathsf{s}.\mathsf{s} = \sigma_1.\mathsf{s}_1.\mathsf{s}_1 * \sigma_2.\mathsf{s}_2.\mathsf{s}_2 \in a * b\,.$$

Consider $\sigma.\mathsf{s} \in a \cap b$. Since a is frameable, we have $\sigma.\mathsf{s}.\mathsf{s} \in a$. With the same argument, $\sigma.\mathsf{s}.\mathsf{s} \in b$. Hence, $\sigma.\mathsf{s}.\mathsf{s} \in a \cap b$.
The proof for union is the same. □

Proof of Lemma 14. We proceed by case analysis.

Case $-\!\!*$*:* "$\subseteq$" Let $\sigma.\mathsf{s} \in \_(p \mathrel{-\!\!*} q)$. To show $\sigma.\mathsf{s} \in \_p \mathrel{-\!\!*} \_q$, consider a computation $\tau.\mathsf{t} \in \_p$ with $\sigma.\mathsf{s} \mathbin{\#} \tau.\mathsf{t}$. Note that definedness of the multiplication guarantees this shape. We have to show $\sigma.\mathsf{s} * \tau.\mathsf{t} = (\sigma * \tau).(\mathsf{s} * \mathsf{t}) \in \_q$. Since $\sigma.\mathsf{s} \in \_(p \mathrel{-\!\!*} q)$, we have $\mathsf{s} \in p \mathrel{-\!\!*} q$. Since $\tau.\mathsf{t} \in \_p$, we have $\mathsf{t} \in p$. Hence, $\mathsf{s} * \mathsf{t} \in q$. This yields $(\sigma * \tau).(\mathsf{s} * \mathsf{t}) \in \_q$.

"$\supseteq$" Let $\sigma.\mathsf{s} \in \_p \mathrel{-\!\!*} \_q$. To show $\sigma.\mathsf{s} \in \_(p \mathrel{-\!\!*} q)$, we have to show $\mathsf{s} \in p \mathrel{-\!\!*} q$. Consider $\mathsf{t} \in p$ with $\mathsf{t} \mathbin{\#} \mathsf{s}$. Let $\mathbf{1}_\sigma$ be the unique computation in $\mathsf{emp}^{|\sigma|}$ such that $\mathbf{1}_\sigma \mathbin{\#} \sigma$. The computation $\mathbf{1}_\sigma.\mathsf{t}$ is in $\_p$. Moreover, since $\mathbf{1}_\sigma \mathbin{\#} \sigma$ and $\mathsf{t} \mathbin{\#} \mathsf{s}$, we have $\mathbf{1}_\sigma.\mathsf{t} \mathbin{\#} \sigma.\mathsf{s}$. Hence, by the assumption, $\mathbf{1}_\sigma.\mathsf{t} * \sigma.\mathsf{s} = \sigma.(\mathsf{t} * \mathsf{s}) \in \_q$. This shows $\mathsf{t} * \mathsf{s} \in q$ as required.

Case $*$*:* "$\subseteq$" Consider $\sigma.\mathsf{s} \in \_(p * q)$. Then $\mathsf{s} \in p * q$. This means $\mathsf{s} = \mathsf{s}_1 * \mathsf{s}_2$ with $\mathsf{s}_1 \in p$ and $\mathsf{s}_2 \in q$. Let $\mathbf{1}_\sigma$ be the unique computation in $\mathsf{emp}^{|\sigma|}$ such that $\mathbf{1}_\sigma \mathbin{\#} \sigma$. Then $\sigma.\mathsf{s}_1 \in \_p$ and $\mathbf{1}_\sigma.\mathsf{s}_2 \in \_q$. Moreover, $\sigma.\mathsf{s}_1 \mathbin{\#} \mathbf{1}_\sigma.\mathsf{s}_2$ with $\sigma.\mathsf{s}_1 * \mathbf{1}_\sigma.\mathsf{s}_2 = \sigma.\mathsf{s} \in \_p * \_q$.

"$\supseteq$" Consider $\sigma.\mathrm{s} \in \_p * \_q$. By definition, there are $\sigma_1.\mathrm{s}_1 \in \_p$ as well as $\sigma_2.\mathrm{s}_2 \in \_q$ such that $\sigma_1.\mathrm{s}_1 * \sigma_2.\mathrm{s}_2 = \sigma.(\mathrm{s}_1 * \mathrm{s}_2) = \sigma.\mathrm{s}$. We have $\mathrm{s}_1 \in p$ and $\mathrm{s}_2 \in q$. Hence, $\mathrm{s} \in p * q$. Hence, $\sigma.\mathrm{s} \in \_(p * q)$.

Case $\cap$*:* "$\subseteq$" Consider $\sigma.\mathrm{s} \in \_(p \cap q)$. We get $\mathrm{s} \in p \cap q$, which means $\mathrm{s} \in p$ and $\mathrm{s} \in q$. Then $\sigma.\mathrm{s} \in \_p$ and $\sigma.\mathrm{s} \in \_q$. This means $\sigma.\mathrm{s} \in \_p \cap \_q$.

"$\supseteq$" Consider $\sigma.\mathrm{s} \in \_p \cap \_q$. Then $\sigma.\mathrm{s} \in \_p$ and $\sigma.\mathrm{s} \in \_q$. Then $\mathrm{s} \in p$ and $\mathrm{s} \in q$, hence $\mathrm{s} \in p \cap q$. Hence, $\sigma.\mathrm{s} \in \_(p \cap q)$.

Case $\overline{\bullet}$*:* Consider $\sigma.\mathrm{s}$. We have $\sigma.\mathrm{s} \in \_(\overline{p})$ iff $\mathrm{s} \in \overline{p}$ iff $\mathrm{s} \notin p$ iff $\sigma.\mathrm{s} \notin \_p$ iff $\sigma.\mathrm{s} \in \overline{\_p}$.

Case $\cup$*:* Follows from $\cap$ and $\overline{\bullet}$.

Case false: We have $\mathit{false} = \emptyset = \Sigma^*.\emptyset = \_\mathit{false}$.

Case true: We have $\mathit{true} = \Sigma^+ = \Sigma^*.\Sigma = \_\mathit{true}$.

Case inclusion: "$\Rightarrow$" Assume $\_p \subseteq \_q$. Let $\mathrm{s} \in p$ be some state. The state is also a computation in $\_p$. As $\_p \subseteq \_q$, we have $\mathrm{s} \in \_q$, which means $\mathrm{s} \in q$.

"$\Leftarrow$" Assume $p \subseteq q$. Consider a computation $\sigma.\mathrm{s} \in \_p$. Then we have $\mathrm{s} \in p$, and hence $\mathrm{s} \in q$. Hence, $\sigma.\mathrm{s} \in \_q$. □

Proof of Lemma 15. We proceed by case analysis.

Case $\_p \subseteq \Diamond p$*:* We have $\_p = \Sigma^*.p \subseteq \Sigma^*.p.\Sigma^* = \Diamond p$.

Case $\Diamond(p * q) \subseteq \Diamond p * \Diamond q$*:* Consider $\sigma \in \Diamond(p * q)$. Then there is a decomposition $\sigma_1.\mathrm{s}.\sigma_2$ with $\mathrm{s} \in p * q$. Then we have $\mathrm{s} = \mathrm{s}_1 * \mathrm{s}_2$ with $\mathrm{s}_1 \in p$ and $\mathrm{s}_2 \in q$. We define $\tau_1 = \sigma_1.\mathrm{s}_1.\sigma_2 \in \Diamond p$. Let $\mathbf{1}_{\sigma_1}$ be the unique computation in $\mathrm{emp}^{|\sigma_1|}$ such that $\mathbf{1}_{\sigma_1} \# \sigma_1$ and define $\mathbf{1}_{\sigma_2}$ similarly for σ_2. We set $\tau_2 = \mathbf{1}_{\sigma_1}.\mathrm{s}_2.\mathbf{1}_{\sigma_2} \in \Diamond q$. Then $\sigma = \tau_1 * \tau_2 \in \Diamond p * \Diamond q$.

Case $\Diamond p \twoheadrightarrow \Diamond q \subseteq \Diamond(p \twoheadrightarrow q)$*:* Consider $\sigma \in \Diamond p \twoheadrightarrow \Diamond q$. Towards a contradiction, assume $\sigma \notin \Diamond(p \twoheadrightarrow q)$. Then for every decomposition $\sigma = \sigma_1.\mathrm{s}.\sigma_2$ we have $\mathrm{s} \notin p \twoheadrightarrow q$. This means there is $\mathrm{t}_\mathrm{s} \in p$ with $\mathrm{t}_\mathrm{s} \# \mathrm{s}$ but $\mathrm{s} * \mathrm{t}_\mathrm{s} \notin q$. Let τ be the computation consisting of all such t_s (in the right order). Then $\tau \in \Diamond p$ and $\tau \# \sigma$. Hence, $\tau * \sigma \in \Diamond q$, because $\sigma \in \Diamond p \twoheadrightarrow \Diamond q$. This, however, contradicts the construction of τ.

Case $\Diamond(p \cap q) \subseteq \Diamond p \cap \Diamond q$*:* We have $p \cap q \subseteq p$ and $p \cap q \subseteq q$. Hence, $\Diamond(p \cap q) \subseteq \Diamond p$ and $\Diamond(p \cap q) \subseteq \Diamond q$ by the equivalence for inclusion. Hence, $\Diamond(p \cap q) \subseteq \Diamond p \cap \Diamond q$.

Case $\Diamond(p \cup q) = \Diamond p \cup \Diamond q$*:* "$\subseteq$" Consider $\sigma \in \Diamond(p \cup q)$. Then $\sigma = \sigma_1.\mathrm{s}.\sigma_2$ with $\mathrm{s} \in p \cup q$, say $\mathrm{s} \in p$. Then $\sigma = \sigma_1.\mathrm{s}.\sigma_2 \in \Diamond p$.
"$\supseteq$" We get $\Diamond(p \cup q) \supseteq \Diamond p$ from $p \subseteq p \cup q$ and the equivalence for inclusion.

Case $\mathit{true} * \Diamond p = \Diamond(p * \mathit{true})$*:* "$\subseteq$" Consider $\tau * \sigma \in \mathit{true} * \Diamond p$. Then, $\tau = \tau_1.\mathrm{t}.\tau_2$ and $\sigma = \sigma_1.\mathrm{s}.\sigma_2$ with $|\tau_1| = |\sigma_1|$, $|\tau_2| = |\sigma_2|$, and $\mathrm{s} \in p$. Consequently, we have $\mathrm{t} * \mathrm{s} \in \mathit{true} * p$. Hence, we arrive at $\tau * \sigma = (\tau_1 * \sigma_1).(\mathrm{t} * \mathrm{s}).(\tau_2 * \sigma_2) \in \Diamond(\mathit{true} * p)$.
"$\supseteq$" Consider $\sigma \in \Diamond(p * \mathit{true})$. Then, $\sigma = \sigma_1.\mathrm{s}.\sigma_2$ with $\mathrm{s} \in p * \mathit{true}$. Then, we get $\mathrm{s} = \mathrm{s}_1 * \mathrm{s}_2$ with $\mathrm{s}_1 \in p$. Then, $\sigma_1.\mathrm{s}_1.\sigma_2 \in \Diamond p$. Let $\mathbf{1}_{\sigma_1}$ be the unique computation in $\mathrm{emp}^{|\sigma_1|}$ such that $\mathbf{1}_{\sigma_1} \# \sigma_1$ and define $\mathbf{1}_{\sigma_2}$ similarly for σ_2. Then $\mathbf{1}_{\sigma_1}.\mathrm{s}_2.\mathbf{1}_{\sigma_2} \in \mathit{true}$ is such that $\sigma_1.\mathrm{s}_1.\sigma_2 \# \mathbf{1}_{\sigma_1}.\mathrm{s}_2.\mathbf{1}_{\sigma_2}$. We have $\sigma_1.\mathrm{s}_1.\sigma_2 * \mathbf{1}_{\sigma_1}.\mathrm{s}_2.\mathbf{1}_{\sigma_2} = \sigma \in \Diamond p * \mathit{true}$.

The remaining cases are similar to the proof of Lemma 14. □

Proof of Lemma 16. *Now:* "$\subseteq$" Consider some $\sigma.\mathsf{s}_1 \in wp(\mathtt{com}, \_p)$, which means that we have $[\![\![\mathtt{com}]\!]\!](\sigma.\mathsf{s}_1) \subseteq \_p$. Hence, for all $\sigma.\mathsf{s}_1.\mathsf{s}_2 \in [\![\![\mathtt{com}]\!]\!](\sigma.\mathsf{s}_1)$ we have $\mathsf{s}_2 \in p$. By definition, $\mathsf{s} \in [\![\mathtt{com}]\!](\mathsf{s}_1)$ implies $\sigma.\mathsf{s}_1.\mathsf{s} \in [\![\![\mathtt{com}]\!]\!](\sigma.\mathsf{s}_1)$. Hence, $[\![\mathtt{com}]\!](\mathsf{s}_1) \subseteq p$, which means $\mathsf{s}_1 \in wp(\mathtt{com}, p)$. Hence, $\sigma.\mathsf{s}_1 \in \_wp(\mathtt{com}, p)$.

"$\supseteq$" For the reverse inclusion, consider $\sigma.\mathsf{s}_1 \in \_wp(\mathtt{com}, p)$. Then $\mathsf{s}_1 \in wp(\mathtt{com}, p)$, which means $[\![\mathtt{com}]\!](\mathsf{s}_1) \subseteq p$. Consider a computation $\sigma.\mathsf{s}_1.\mathsf{s}_2 \in [\![\![\mathtt{com}]\!]\!](\sigma.\mathsf{s}_1)$. By definition, $\mathsf{s}_2 \in [\![\mathtt{com}]\!](\mathsf{s}_1)$. Since $[\![\mathtt{com}]\!](\mathsf{s}_1) \subseteq p$, we get $\sigma.\mathsf{s}_1.\mathsf{s}_2 \in \_p$. This shows $[\![\![\mathtt{com}]\!]\!](\sigma.\mathsf{s}_1) \subseteq \_p$, which means $\sigma.\mathsf{s}_1 \in wp(\mathtt{com}, \_p)$.

Past: "$\subseteq$" Consider $\sigma.\mathsf{s}_1 \in wp(\mathtt{com}, \diamonddot p)$, which means that we have $[\![\![\mathtt{com}]\!]\!](\sigma.\mathsf{s}_1) \subseteq \diamonddot p$. Hence, for all $\sigma.\mathsf{s}_1.\mathsf{s}_2 \in [\![\![\mathtt{com}]\!]\!](\sigma.\mathsf{s}_1)$ we have $\sigma.\mathsf{s}_1.\mathsf{s}_2 \in \diamonddot p$. There are two cases.
Case 1: A state from $\sigma.\mathsf{s}_1$ belongs to p. Then $\sigma.\mathsf{s}_1 \in \diamonddot p$.
Case 2: No state from $\sigma.\mathsf{s}_1$ belongs to p. Then for all $\sigma.\mathsf{s}_1.\mathsf{s}_2 \in [\![\![\mathtt{com}]\!]\!](\sigma.\mathsf{s}_1)$ we have $\mathsf{s}_2 \in p$. This means $[\![\![\mathtt{com}]\!]\!](\sigma.\mathsf{s}_1) \subseteq \_p$. Hence, $\sigma.\mathsf{s}_1 \in wp(\mathtt{com}, \_p)$.

"$\supseteq$" For the reverse inclusion, consider $\sigma.\mathsf{s}_1 \in \diamonddot p$. Then for all $\sigma.\mathsf{s}_1.\mathsf{s}_2 \in [\![\![\mathtt{com}]\!]\!](\sigma.\mathsf{s}_1)$ we have $\sigma.\mathsf{s}_1.\mathsf{s}_2 \in \diamonddot p$. Hence, $\sigma.\mathsf{s}_1 \in wp(\mathtt{com}, \diamonddot p)$.
Consider $\sigma.\mathsf{s}_1 \in wp(\mathtt{com}, \_p)$. Then for all $\sigma.\mathsf{s}_1.\mathsf{s}_2 \in [\![\![\mathtt{com}]\!]\!](\sigma.\mathsf{s}_1)$ we have $\sigma.\mathsf{s}_1.\mathsf{s}_2 \in \_p$. Since $\_p \subseteq \diamonddot p$, we get $\sigma.\mathsf{s}_1 \in wp(\mathtt{com}, \diamonddot p)$. □

Proof of Lemma 18. Rule h-intro is trivial. Rule h-infer is a consequence of Lemma 15. So, consider Rule h-hindsight. Let $(\sigma, i) \in \_p * \diamonddot q$. This means there are $(\sigma_1, i) \in \_p$ and $(\sigma_2, i) \in \diamonddot q$ so that $\sigma = \sigma_1 * \sigma_2$. Since $\_p$ is pure, we know that $\sigma_1 \in \mathsf{emp}^+$. This yields $\sigma = \sigma_2$ and thus $(\sigma, i) \in \diamonddot q$. Again since $\_p$ is pure, we have $p \subseteq \mathsf{emp}$. Hence, we obtain $q \subseteq p * q$ and thus $\diamonddot q \subseteq \diamonddot(p * q)$ by Lemma 15. □

Recall that a predicate i is *intuitionistic*, if $i * \mathit{true} = i$. Intuitionism propagates to the computation predicates, where *true* is defined to be the set of all computations.

Lemma 59. *If i is an intuitionistic state predicate, then $\_i$ and $\diamonddot i$ are intuitionistic computation predicates.*

Proof of Lemma 59. Consider $\_i$. We have $\_i * \mathit{true} = \_i * \_\mathit{true} = \_(i * \mathit{true}) = \_i$. The first two equalities are by Lemma 14, the last additionally uses the fact that i is intuitionistic. The case of $\diamonddot i$ is similar. □

H PROOFS OF SECTION C

Proof of Lemma 24. The laws of commutativity and units follow immediately from the definition of composition. For associativity we need to show that $c_2 \# c_3$ and $c_1 \#(c_2 * c_3)$ if and only if $c_1 \# c_2$ and $(c_1 * c_2) \# c_3$. We prove the direction from left to right. The reverse direction holds by symmetry of definedness.

Since $c_2 \# c_3$, we have $(c_2 \uplus c_3).\mathit{flow} \geq c_2.\mathit{flow} \uplus c_3.\mathit{flow}$. Furthermore, by $c_1 \#(c_2 * c_3)$ we have

$$(c_1 \uplus (c_2 * c_3)).\mathit{flow} \geq c_1.\mathit{flow} \uplus (c_2 * c_3).\mathit{flow}\ .$$

Note that $(c_1 \uplus (c_2 * c_3)).\mathit{flow} = (c_1 \uplus c_2 \uplus c_3).\mathit{flow}$.

To show $c_1 \# c_2$, we have to argue that

$$(c_1 \uplus c_2).\mathit{flow} \geq c_1.\mathit{flow} \uplus c_2.\mathit{flow}\ .$$

To see this, note that

$$(c_1 \uplus c_2).\mathit{flow} \geq (c_1 \uplus c_2 \uplus c_3).\mathit{flow}|_{(c_1 \uplus c_2).X} \geq c_1.\mathit{flow} \uplus c_2.\mathit{flow}\ .$$

The latter inequality is by the assumptions. For the former inequality, we note that the fixed point iteration for $(c_1 \uplus c_2).\mathit{flow}$ starts with a contribution from c_3 (given as inflow) that the iteration for $(c_1 \uplus c_2 \uplus c_3).\mathit{flow}$ only receives when reaching the fixed point. By monotonicity, every fixed point approximant to the left is then larger than the corresponding approximant to the right, and so is the fixed point. For $(c_1 * c_2) \# c_3$, we note that

$$((c_1 * c_2) \uplus c_3).\mathit{flow} = (c_1 \uplus c_2 \uplus c_3).\mathit{flow} \geq c_1.\mathit{flow} \uplus c_2.\mathit{flow} \uplus c_3.\mathit{flow}$$
$$\geq (c_1 \uplus c_2).\mathit{flow} \uplus c_3.\mathit{flow}\ .$$

The first inequality is by the above assumptions. The second always holds, as remarked above. □

Proof of Lemma 29. (i): For $c_2.X \cap c.X = \emptyset$, we use the fact that $c_1.X \cap c.X = \emptyset$ follows from $c_1 \# c$ and $c_1.X = c_2.X$.

To see that the outflow of c_2 matches the inflow of c, we consider $x \in c_2.X$ and $y \in c.X$ and reason as follows:

$$c_2.\mathit{out}(x, y) = c_1.\mathit{out}(x, y) = c.\mathit{in}(x, y).$$

The former equality follows from $c_2.\mathit{in} = c_1.\mathit{in}$ together with $\mathit{tf}(c_1) =_{c_1.\mathit{in}} \mathit{tf}(c_2)$. The second equality is by $c_1 \# c$.

The inflow is preserved by the assumption, hence we have the following:

$$c.\mathit{out}(y, x) = c_1.\mathit{in}(y, x) = c_2.\mathit{in}(y, x)\ .$$

It remains to show $(c_2 \uplus c).\mathit{flow} = c_2.\mathit{flow} \uplus c.\mathit{flow}$. We use Bekic's lemma. Define the target pairing of two functions $f : A \to B$ and $g : A \to C$ over the same domain A as the function $\langle f, g\rangle : A \to B \times C$ with $\langle f, g\rangle(a) \triangleq (f(a), g(a))$. We compute the flow of $c_2 \uplus c$ as the least fixed point of a target pairing $\langle f, g\rangle$ with

$$f : ((c_2.X \uplus c.X) \to \mathbb{M}) \to c_2.X \to \mathbb{M}$$
$$g : ((c_2.X \uplus c.X) \to \mathbb{M}) \to c.X \to \mathbb{M}\ .$$

Function f updates the flow of the nodes in c_2 depending on the flow in/inflow from c. Function g is responsable for the flow of the nodes in c. The inflow from the nodes outside $c_2 \uplus c$ is constant. The definition guarantees $(c_2 \uplus c).\mathit{flow} = \mathit{lfp}(\langle f, g\rangle)$.

We curry the former function,

$$f :\ (c.X \to \mathbb{M}) \to (c_2.X \to \mathbb{M}) \to c_2.X \to \mathbb{M}\ ,$$

and obtain, for every $\mathit{cval} : c.X \to \mathbb{M}$, the function

$$f(\mathit{cval}) :\ (c_2.X \to \mathbb{M}) \to c_2.X \to \mathbb{M}\ .$$

This function is still monotonic and therefore has a least fixed point. Hence, the function

$$f^\dagger : (c.X \to \mathbb{M}) \to c_2.X \to \mathbb{M}$$

mapping valuation $\mathit{cval} : c.X \to \mathbb{M}$ to the least fixed point $\mathit{lfp}(f(\mathit{cval}))$ is well-defined.

Bekić's lemma [Bekic 1984] tells us how to compute least fixed points of target pairings like $\langle f, g\rangle$ above by successive elimination of the variables. We first determine $f^\dagger$, which is a function in $c.X \to \mathbb{M}$. We plug this function into g to obtain a function solely in $c.X \to \mathbb{M}$. To be precise, since g expects a function from $(c_2.X \uplus c.X) \to \mathbb{M}$, we pair $f^\dagger$ with $\mathit{id} = \mathit{id}_{c.X \to \mathbb{M}}$ and obtain

$$\langle f^\dagger, \mathit{id}\rangle : (c.X \to \mathbb{M}) \to (c_2.X \uplus c.X) \to \mathbb{M}\ .$$

We compose this function with g and get

$$g \circ \langle f^\dagger, \mathit{id}\rangle : (c.X \to \mathbb{M}) \to c.X \to \mathbb{M}\ .$$

We compute the least fixed point of this composition to obtain the values of the least fixed point of interest on $c.X$. For the values on $c_2.X$, we reinsert the $c.X$-values into $f^\dagger$. Bekić's lemma guarantees the correctness of this successive elimination procedure:

$$\begin{aligned} \mathit{lfp}(\langle f, g\rangle) &= (f^\dagger(\mathit{cval}), \mathit{cval}) \\ \text{with} \qquad \mathit{cval} &= \mathit{lfp}(g \circ \langle f^\dagger, \mathit{id}\rangle)\ . \end{aligned}$$

To conclude the proof, we recall that $\mathit{tf}(c_2) =_{c_1.\mathit{in}} \mathit{tf}(c_1)$. Moreover, for all $y \in c.X$ and $x \in c_1.X$ we have

$$c_1.\mathit{in}(y, x) = c.\mathit{out}(y, x) = c.E_{(y,x)}(c.\mathit{flow}(x))\ .$$

Together with monotonicity of the edge functions, this implies $g \circ \langle f^\dagger, \mathit{id}\rangle =_{c.\mathit{flow}} g \circ \langle h^\dagger, \mathit{id}\rangle$. Here, $h : ((c_2.X \uplus c.X) \to \mathbb{M}) \to c_2.X \to \mathbb{M}$ is the transformer derived from c_1 in the same way f was derived from c_2. We thus have for all $x \in c.X$:

$$\begin{aligned} & && (c_2 \uplus c).\mathit{flow}(x) \\ (\text{ Definition } \mathit{flow}, f, g\ ) &&=\ & \mathit{lfp}(\langle f, g\rangle)(x) \\ (\text{ Bekić's lemma, } x \in c.X\ ) &&=\ & \mathit{lfp}(g \circ \langle f^\dagger, \mathit{id}\rangle)(x) \\ (\ \mathit{tf}(c_2) =_{c_1.\mathit{in}} \mathit{tf}(c_1), \text{ see above }) &&=\ & \mathit{lfp}(g \circ \langle h^\dagger, \mathit{id}\rangle)(x) \\ (\text{ Bekić's lemma, } x \in c.X\ ) &&=\ & \mathit{lfp}(\langle h, g\rangle)(x) \\ (\text{ Definition flow}, f, g\ ) &&=\ & (c_1 \uplus c).\mathit{flow}(x) \\ (\ c_1 \# c,\ x \in c.X\ ) &&=\ & c.\mathit{flow}(x). \end{aligned}$$

We argue that also for nodes $x \in c_2.X$ that we have the equality $(c_2 \uplus c).\mathit{flow}(x) = c_2.\mathit{flow}(x)$, as follows:

$$\begin{aligned} & && (c_2 \uplus c).\mathit{flow}(x) \\ (\text{ Definition } \mathit{flow}, f, g\ ) &&=\ & \mathit{lfp}(\langle f, g\rangle)(x) \\ (\text{ Bekić's lemma, } x \in c_2.X\ ) &&=\ & [f^\dagger(\mathit{cval})](x) \\ (\text{ See above }) &&=\ & [f^\dagger(c.\mathit{flow})](x) \\ (\text{ See below }) &&=\ & c_2.\mathit{flow}(x). \end{aligned}$$

To see the last equality, note that we have $c_2.\mathit{in}(y, x) = c.\mathit{out}(y, x) = c.E_{(y,x)}(c.\mathit{flow}(x))$, for all $y \in c.X$. Hence, $c_2.\mathit{flow}$, which we compute from the inflow, is the least fixed point of f computed with $c.\mathit{flow}$ fixed.

(ii). Follows with a similar but simpler application of Bekić's lemma. □

Proof of Lemma 34. We show the individual cases.

(i): We apply Lemma 29 to derive that $[\![\mathit{up}]\!](c) \# c'$ holds. To see that the preconditions in the lemma are met, note that we have $c \# c'$ by assumption, $[\![\mathit{up}]\!](c).X = c.X$ and $[\![\mathit{up}]\!](c).\mathit{in} = c.\mathit{in}$ because the semantics does not abort, and the application of updates neither changes the set of nodes nor the inflow, and $\mathit{tf}(c) = \mathit{tf}([\![\mathit{up}]\!](c))$ because the update does not abort.

(ii): Since the separating conjunctions on both sides of the equality are defined by (i) resp. the assumption, the sets of nodes and the inflows coincide. The sets of edges coincide, because $\mathrm{dom}(\mathit{up}) \subseteq c.X \times \mathbb{N}$ as the update does not abort on c.

(iii): With the previous argument, we arrive at: $\mathrm{dom}(\mathit{up}) \subseteq (c.X \uplus c'.X) \times \mathbb{N} = (c * c').X \times \mathbb{N}$. Here, disjointness and the separating conjunction rely on $c \# c'$. For the transformer, we use

$$\mathit{tf}((c * c')[\mathit{up}]) = \mathit{tf}([\![\mathit{up}]\!](c) * c') = \mathit{tf}(c * c')\ .$$

The former equality is by (ii). The latter is Lemma 29 and we already argued that the conditions are met.

(iv) : By (iii), we have $[\![up]\!](c * c') = (c * c')[up]$. The desired equality is by (ii). □

Proof of Lemma 28. Since separation algebras are closed under Cartesian products, the product operation on flow graphs is associative and commutative and $(h_\emptyset, c_\emptyset)$ is the neutral element.

We show $(h_\emptyset, c_\emptyset) \in FG$. For the nodes, we have

$$h_\emptyset.X = \emptyset = c_\emptyset.X\ .$$

The empty edge function $c.E = \emptyset$ is trivially induced (the universal quantifier is over an empty set).

We show closedness: if $fg_1, fg_2 \in FG$ and $fg_1 \# fg_2$, then we also have $fg_1 * fg_2 \in FG$. Due to the Cartesian product construction, we have $(fg_1.h * fg_2.h, fg_1.c * fg_2.c) \in HG \times FC$. Note that this uses the fact that $fg_1 \# fg_2$ implies $fg_1.h \# fg_2.h$ and $fg_1.c \# fg_2.c$.

It remains to show that the constraints on nodes and edges hold. For the nodes, we have

$$(fg_1.h * fg_2.h).X = fg_1.h.X \uplus fg_2.h.X = fg_1.c.X \uplus fg_2.c.X = (fg_1.c * fg_2.c).X\ .$$

The first equality is by the definition of products on heap graphs, the second is the assumption $fg_1, fg_2 \in FG$, and the last is the definition of products on flow constraints. For the edge function, we consider $x \in fg_1.h.X$ and $y \in \mathbb{N}$. For $x \in fg_2.h.X$, the reasoning is similar:

$$\begin{aligned}
&\sum_{\substack{\text{psel}\in PSel\\ y=(fg_1.h*fg_2.h).pval(x,\text{psel})}} gen(\text{psel}, (fg_1.h * fg_2.h).dval(x))\\
\overset{(1)}{=}\ &\sum_{\substack{\text{psel}\in PSel\\ y=(fg_1.h.pval\uplus fg_2.h.pval)(x,\text{psel})}} gen(\text{psel}, (fg_1.h.dval \uplus fg_2.h.dval)(x))\\
\overset{(2)}{=}\ &\sum_{\substack{\text{psel}\in PSel\\ y=fg_1.h.pval(x,\text{psel})}} gen(\text{psel}, fg_1.h.dval(x))\\
\overset{(3)}{=}\ &fg_1.c.E(x, y)\\
\overset{(4)}{=}\ &(fg_1.c.E \uplus fg_2.c.E)(x, y)\\
\overset{(5)}{=}\ &(fg_1.c * fg_2.c).E(x, y)\ .
\end{aligned}$$

where the equalties are due to: (1) the definition of products of heap graphs, (2) $x \in fg_1.h.X$ together with disjointness, (3) the fact that fg_1, fg_2 are flow graphs, (4) $x \in fg_1.h.X$ together with union of functions, and (5) the definition of products of flow constraints. □