CoLoSL: Concurrent Local Subjective Logic

Azalea Raad, Jules Villard, and Philippa Gardner

Imperial College London
{azalea,j.villard,pg}@doc.ic.ac.uk

Abstract. A key difficulty in verifying shared-memory concurrent programs is reasoning compositionally about each thread in isolation. Existing verification techniques for fine-grained concurrency typically require reasoning about either the entire shared state or disjoint parts of the shared state, impeding compositionality. This paper introduces the program logic CoLoSL, where each thread is verified with respect to its subjective view of the global shared state. This subjective view describes only that part of the state accessed by the thread. Subjective views may arbitrarily overlap with each other, and expand and contract depending on the resource required by the thread. This flexibility gives rise to small specifications and, hence, more compositional reasoning for concurrent programs. We demonstrate our reasoning on a range of examples, including a concurrent computation of a spanning tree of a graph.

1 Introduction

A key difficulty in verifying properties of fine-grained concurrent programs is being able to reason compositionally about each thread in isolation, even though in reality the correctness of the whole system is the collaborative result of intricately intertwined actions of the threads. Such compositional reasoning is essential for: verifying large concurrent systems, since it allows them to be verified component-wise; verifying library code and incomplete programs, since one does not need to know about the context of execution; and replicating a programmer's intuition about why their implementations are correct, since their informal arguments are typically kept local and do not bring the whole system into the reasoning.

Rely-guarantee [16] and various combinations of rely-guarantee and separation logic reasoning [26,9,10,7,23,3] achieve compositional reasoning for increasingly difficult inter-thread interactions. However, we believe that, despite substantial progress, there are many examples where the specifications and proofs are not as concise as they might be; they are either too coarse or too contrived. We explore a different approach, introducing the program logic CoLoSL, using which we can give small specifications and proofs which correspond to the programmer's intuition of what shared resource is required by the thread.

Small specifications were emphasised in the work of O'Hearn, Reynolds and Yang on separation logic [19]. The original separation logic [21,15] achieves local, compositional reasoning for sequential heap-manipulating programs by splitting the heap into disjoint heaplets for describing the local resources required by a program. Compositionality then rests on two powerful mechanisms: a program can be

J. Vitek (Ed.): ESOP 2015, LNCS 9032, pp. 710–735, 2015.
DOI: 10.1007/978-3-662-46669-8_29

specified using only those resources that it actually accesses; and this specification can be simply reused in any context that contains these resources. By making the specification as small as possible, we can ensure that it can be reused in a large set of possible contexts using the *frame rule*. This plays an important role in achieving scalable compositional reasoning, as each block of code can be proved in isolation and its small specification reused in larger contexts.

Concurrent separation logic extends this compositional reasoning to concurrent programs using the *disjoint concurrency rule*, where individual threads use both local resources private to the thread as well as static shared resources, which can be accessed by all threads through critical sections. Since then, there have been many extensions combining rely-guarantee reasoning with ideas from concurrent separation logic to reason about fine-grained concurrency: RGSep [26] introduced local resource and shared disjoint regions which are stable with respect to a static interference relation stating how the current thread and the environment can affect the region; CAP [7] and its extensions [24,23,3] in addition abstract the regions. This work has achieved substantial success; see the related work section for more details. Yet, we believe these approaches are not always able to provide small specifications for concurrent programs, impeding compositionality.

The problem is due to the rigid nature of the shared disjoint regions and their static interference, which limits how we can work with a concurrent data structure. A disjoint region typically either describes the entire data structure such as a linked list, or contains individual components of the data structure such as the nodes of a linked list. However, threads may have shared access to arbitrary parts of the data structure, which cannot be directly expressed in the reasoning. The interference associated with the region is static in that it is defined when the region is created and is fixed throughout its entire lifetime. However, threads having access to parts of a shared region need only know about the interference on those parts. In addition, it is not always possible to predict all future interactions associated with a shared region at the time of creation. Just as the original separation logic uses the frame rule to obtain small specifications for the *local* state, we seek an analogous framing mechanism on both the shared resource and its interference to obtain small specifications of the *shared* state.

For example, consider a linked list consisting of $n+1$ nodes accessed concurrently by n threads where the ith thread requires access to the ith and $(i+1)$st nodes. Current approaches cannot provide a small specification for each thread which captures just these two shared nodes and their interference.[1] Now consider a program whose threads manipulate subgraphs of a graph, such as a recursive concurrent spanning tree algorithm. Current approaches cannot give small specifications capturing just the subgraph manipulated by each thread due to intrinsic, unspecified sharing between subgraphs. Finally, consider a concurrent set implementation. In CoLoSL it is possible to describe the interference associated with a new element as it is added to the set. The interference on new elements need not be the same as before and may

[1] In [13], similar problems were encountered in an attempt to provide small specifications for a doubly-linked list implementation of a concurrent tree.

only be known at the time they are added. Existing approaches cannot accommodate such dynamic interference relations.

This paper introduces the program logic CoLoSL, which stands for Concurrent Local Subjective Logic. CoLoSL's semantic model is based on one global shared state, and each thread is verified with respect to its partial *subjective view* of this state. Each subjective (personalised) view comprises an assertion which describes *parts* of the shared global resource used by a thread, and an interference relation which describes how the thread and the environment may affect these parts. Subjective views may arbitrarily overlap with each other, with both their resources and interference expanding and contracting in accordance with what is required by the thread. Interestingly, this sometimes requires rewriting the interference relation so that the interference on the smaller state captures the same information as the interference on the bigger state. This expansion and contraction of subjective views provides small specifications for individual threads and local reasoning about a thread's shared state.

We demonstrate CoLoSL reasoning on a range of examples. The first example in §2 is Dijkstra's token ring mutual exclusion algorithm [5]. Regardless of the size of the ring, we are able to give a small specification to each thread in isolation such that each proof only mentions resources associated with two of its neighbours. This means that the proof can be largely reused when the implementation of the ring is changed to allow dynamic spawning of new participants. In §4, we study two further examples. The first is a concurrent spanning tree algorithm for graphs, where threads are recursively spawned on potentially overlapping subgraphs. We demonstrate that the flexible, overlapping subjective views of CoLoSL are just what we need. The second is a concurrent set module implemented using a hand-over-hand list-locking algorithm. Our CoLoSL reasoning for this set module improves on CAP reasoning [7] in that our small specifications can be dynamically extended to include other behaviour in future whereas the static CAP interference must predict all behaviour from the start.

Most of the technical details have been left out due to space constraints, and are provided in the accompanying technical report [20].

Related work. Jones' rely-guarantee reasoning [16] provided a breakthrough in compositional reasoning about concurrent programs. It described permitted interferences between threads using global rely and guarantee relations on one global shared state. This work is compositional in the sense that the guarantee of one thread is the rely of another. However, O'Hearn [18] demonstrated that, for concurrent reasoning to scale, it is important to work with small specifications based on local state and shared state rather than working with one global state. In particular, he introduced concurrent separation logic based on thread-local state and static invariants for the shared state.

Since then, there has been much recent work on compositional reasoning about fine-grained concurrent algorithms. RGSep [26] combined rely-guarantee reasoning with separation logic, with the state split into thread-local state and disjoint shared regions, and global rely-guarantee relations providing the interference on these regions. Deny-guarantee extends rely-guarantee and RGSep with permis-

sions that can turn pieces of interference on or off during the proof of a program, as would be typically needed in a programming language with fork and join constructs instead of statically-scoped parallel composition. This has influenced the capabilities of CAP. CAP reasoning [7] increased compositionality by, instead of the global rely-guarantee relations, having static interference relations associated with the disjoint shared regions and capabilities in the local state for dynamically controlling the permitted interference, and concurrent abstract predicates for hiding implementation details. Several program logics have adapted this work to incorporate more abstraction [25], higher-order features [23] and more flexible capabilities [3]. We believe that all this work has limited compositional reasoning in the sense that it is only possible to frame off local state and unused shared regions. It is not possible to frame within regions nor frame inside the interference relations. (It is, of course, possible to weaken the region assertion by the logical implication $P * Q \Rightarrow P * \mathsf{true}$, but this is not the same as being able to shrink regions *and* their interference to just work with P.)

Meanwhile, a different breakthrough in compositionality came from Feng's local rely-guarantee (LRG) reasoning [10]. The LRG model comprises one global shared state, with assertions describing disjoint but flexible parts of this state and rely-guarantee relations determining how the thread and environment can affect these partial states. With LRG, it is possible to frame off disjoint parts of shared state and their associated disjoint rely-guarantee relations, but, as noted by its author, the strong disjointness restrictions make this approach only applicable for disjoint modules in the code. We took significant inspiration from LRG, combining the flexibility of the LRG approach with our subjective views to reason locally about overlapping shared states.

Finally, Fine-grained Concurrent Separation Logic (FCSL) [17] of Nanevski *et al.* explores a different notion of region called *concurroids*. Like the regions in CAP, concurroids describe disjoint pieces of shared state together with their static interference. It is therefore not possible to frame within concurroids. The term "subjectivity" in FCSL refers to the fact that, unlike CAP, concurroids have three parts: the "joint" part; and the disjoint "self" and "other" parts. Although FCSL did not influence CoLoSL, it does highlight an interesting point regarding resource transfer between regions: in CAP, communication between regions is achieved indirectly via the local state; in FCSL, communication between concurroids is achieved directly through dangling external transitions; and, in CoLoSL, communication between compatible subjective views can be achieved by merging the two views.

2 Informal Development

We sketch a proof of an implementation of Dijkstra's token ring mutual exclusion algorithm, which pioneered *self-stabilising* distributed algorithms [5]. Our proof highlights the main reasoning principles of CoLoSL and results in *small specifications* for each participant in the ring. Besides their concision, we show how small specifications are robust against non-trivial changes to the program.

The algorithm assumes a network of $n+1$ machines arranged in a ring, with a designated *master* machine and n slave machines. Each machine maintains a

$$\boxed{\boxed{P * Q}_I} \Rightarrow \boxed{\boxed{P}_I}_I \quad \text{(Forget)} \qquad \boxed{\boxed{P}_{I_1} * \boxed{Q}_{I_2}} \Rightarrow \boxed{\boxed{P \circledast Q}_{I_1 \cup I_2}}_{I_1 \cup I_2} \quad \text{(Merge)}$$

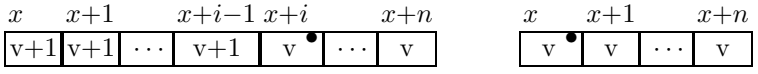
$$(P \Rightarrow Q) \Rightarrow \boxed{\boxed{P}_I}_I \Rightarrow \boxed{\boxed{Q}_I}_I \quad \text{(Weaken)} \qquad I \sqsubseteq^P I' \text{ implies } \boxed{\boxed{P}_I}_I \Rightarrow \boxed{\boxed{P}_{I'}}_{I'} \quad \text{(Shift)}$$

$$\boxed{\boxed{P}_I}_I \Rightarrow \boxed{\boxed{P}_I}_I * \boxed{\boxed{P}_I}_I \quad \text{(Copy)} \qquad P \circledcirc I \text{ implies } P \Rrightarrow \exists \mathsf{A}, \mathsf{A}'.\, [\mathsf{A}] * \boxed{\boxed{P * [\mathsf{A}']}_I}_I \quad \text{(Extend)}$$

$$\frac{\{P_1\}\ \mathbb{C}_1\ \{Q_1\} \qquad \{P_2\}\ \mathbb{C}_2\ \{Q_2\}}{\{P_1 * P_2\}\ \mathbb{C}_1 \parallel \mathbb{C}_2\ \{Q_1 * Q_2\}}\ \textsc{Par} \qquad \frac{P \Rrightarrow P' \quad \vdash \{P'\}\ \mathbb{C}\ \{Q'\} \quad Q' \Rrightarrow Q}{\vdash \{P\}\ \mathbb{C}\ \{Q\}}\ \textsc{Consq}$$

Fig. 1. Main reasoning principles and rules of CoLoSL

local counter and has access to the value of its left neighbour's counter; the state of the system consists of all $n{+}1$ counters. Starting in an arbitrary state, the network eventually stabilises to legitimate states [4], with the following global property: in the ith legitimate state, machines 0 to $i{-}1$ (machine 0 being the master) have some value $v{+}1$, and all others have value v. In the ith legitimate state, only the ith machine (indicated by • below) can make progress: it increments its counter by 1 and takes the system to the next legitimate state ($i{+}1 \bmod n{+}1$). For a ring at address x, the ith legitimate state is depicted below (for both $i > 0$ and $i = 0$).



In [20], we outline a proof of the token ring's self-stabilisation phase, which shows that the system always converges to a legitimate state. We also provide a proof of the token ring as a mutual exclusion mechanism, where a machine holding • may gain ownership of a shared resource. For simplicity, here we focus on the case where the token ring is in the 0th legitimate state, with all counters holding value 0. Our proof makes use of the CoLoSL principles laid out in Fig. 1, together with the usual concurrency rule of separation logic [18] and the rule of semantic consequence from the views framework [6]. We introduce them informally as needed, and discuss them in more detail in §3.

CoLoSL introduces a new assertion $\boxed{\boxed{P}_I}_I$ called a *subjective view*, which comprises a *subjective assertion* P describing a *part* of the global shared state and an *interference assertion* I characterising how this partial shared state may be changed by the thread or the environment. Similar to the interference assertions of CAP [7], I declares actions of the form $[\mathsf{a}] : Q \rightsquigarrow R$, where a thread in possession of the $[\mathsf{a}]$ capability in its local state may carry out its transition and update parts of the shared state satisfying Q to a state satisfying R. Assertions in Hoare triples must be *stable* with respect to the interference from the environment: that is, robust with respect to the interference assertion I.

Consider the program ring(x) defined in Fig. 2 which represents a token ring with $n{+}1$ machines. It is written in pseudo-code resembling C with additional constructs for concurrency: atomic sections $\langle \_ \rangle$ which declare that code behaves atomically; and parallel composition $\_ \parallel \_$ which spawns threads then waits until they complete. In our example, the $n{+}1$ threads run in parallel until all counters

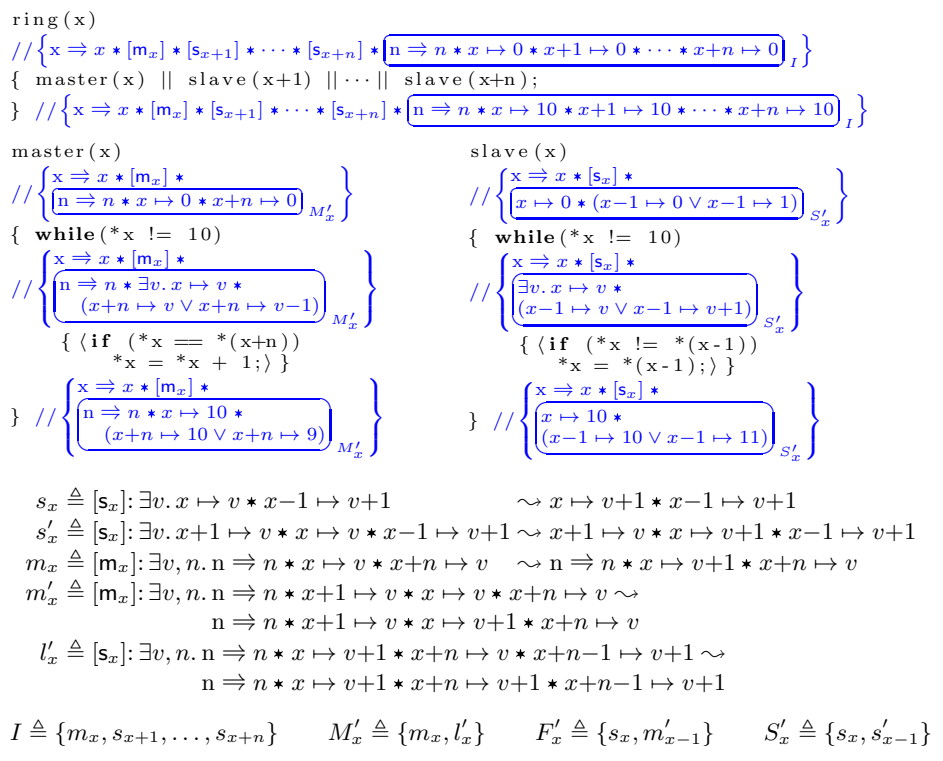


Fig. 2. A proof sketch of the token ring in CoLoSL. Assertions in lines starting with // describe the local state and the subjective shared state at the relevant program points. The proof of slave(x) applies to all slaves expect the first one (called the foreman) in the parallel composition, where S'_x is replaced by F'_x.

reach value 10. While the implementation of all the slave threads are identical, we shall see that the proof of the first slave in the ring (at $x{+}1$) is slightly different from the others. We henceforth refer to the first slave thread as the *foreman*. Let us proceed with the proof of the other slaves.

Proof of the slaves. Let us temporarily forget about the proof outline of Fig. 2 and attempt to prove slave(x) in isolation, in the spirit of local reasoning. Since slave(x) inspects the value of its counter pointed to by x and compares it against the counter at x-1 (its left neighbour in the ring), a tempting precondition for slave(x) would be one describing just these two locations, e.g.

$$\mathrm{x} \Rightarrow x * [\mathsf{s}_x] * \boxed{x \mapsto 0 * (x{-}1 \mapsto 0 \vee x{-}1 \mapsto 1)}_{S_x} \qquad S_x = \{s_x, s_{x-1}\}$$

The above assertion comprises: a) a variable assertion stating that the thread locally owns variable x with value x (using the variables-as-resource model [1]); b) a *capability* $[\mathsf{s}_x]$ that allows it to perform the associated s_x action (see below); and c) a subjective view of the shared state: x points to 0, and $x{-}1$, its left

neighbour, points to either 0 or 1 since it might already have incremented its own counter. The $*$ connective used between assertions is that of separation logic and means that the assertions describe *disjoint* pieces of state. The interference assertion associated with the subjective view is captured by S_x and consists of two actions: s_x and s_{x-1}, where s_x represents an increment of the contents of x under the condition that its value is one less than the value at address $x-1$ (see Fig. 2). Since the current thread owns $[\mathsf{s}_x]$ locally, only it can perform s_x. On the other hand, the capability $[\mathsf{s}_{x-1}]$ is not locally owned, thus the environment could potentially perform the associated action whenever its precondition (on the left-hand side of $\rightsquigarrow$) is satisfied. Upon closer inspection, since this subjective view says nothing of the value of the cell at address $x-2$, s_{x-1} could potentially *always* fire. The assertion is thus not *stable*: nothing prevents the counter at $x-1$ from incrementing beyond value 1. A weaker, stable assertion is thus:

$$\mathrm{x} \Rrightarrow x * [\mathsf{s}_x] * \boxed{\exists v.\, x \mapsto 0 * x{-}1 \mapsto v}_{S_x}$$

Fortunately, we can do better and obtain a stronger small precondition. Let us first step back and think again at the level of the whole algorithm. As the programmer knows, the situation above cannot happen as $x-2$ can only be at most one ahead of x itself. We can thus replace S_x by S'_x and give a stronger stable precondition that captures just what we want, as in Fig. 2. The proof of slave(x) is now relatively straightforward. By inspection (or using the rules of §3.3), the invariant of the while loop and the postcondition are also stable. The atomic section temporarily "opens the box" to perform action s_x then "closes back the box", and preserves the invariant. The final postcondition of slave(x) follows from the invariant and the boolean expression of the loop.

Proof of the master and foreman. The proof sketches of the master and foreman threads given in Fig. 2 are analogous. As the first slave, the foreman has to account for interference from the master instead of another slave. Moreover, the master and its associated action m_x have access to variable n holding the current size of the ring, since they depend on the value of the counter at address $x + n$.

Proof of the ring. The precondition of ring(x) states that it owns all capabilities, which will be distributed amongst the $n+1$ threads; the global variable $\mathrm{n} \Rrightarrow n$ is shared, as are all $n+1$ counters, initialised to 0. The interference I associated with the subjective view consists of the actions of the $n+1$ threads. Because ring(x) has a global view of the state of the ring (and moreover all capabilities are held locally), the s_{x+i} actions are enough to guarantee stability.

Let us write $\boxed{P}_I$ for this initial subjective view. This assertion may be freely duplicated using the COPY principle of Fig. 1 and each thread is given a copy together with the appropriate capability using the usual PAR rule of concurrent separation logic. For instance, the thread running slave(x+i), for $i > 1$, gets $[\mathsf{s}_{x+i}] * \boxed{P}_I$. This assertion does not match the precondition of slave(x+i) just yet. Using the principles of Fig. 1, we can weaken the assertion as such:

$$\boxed{P}_I \overset{(\textsc{Shift})}{\Rightarrow} \boxed{P}_{(I\setminus\{s_{x+i-1}\})\cup\{s'_{x+i-1}\}} \overset{(\textsc{Forget})}{\Rightarrow} \boxed{x{+}i{-}1 \mapsto 0 * x{+}i \mapsto 0}_{(I\setminus\{s_{x+i-1}\})\cup\{s'_{x+i-1}\}} \overset{(\textsc{Shift})}{\Rightarrow} \boxed{x{+}i{-}1 \mapsto 0 * x{+}i \mapsto 0}_{\{s'_{x+i-1},s_{x+i}\}} \overset{(\textsc{Weaken})}{\Rightarrow} \boxed{x{+}i \mapsto 0 * (x{+}i{-}1 \mapsto 0 \vee x{+}i{-}1 \mapsto 1)}_{\{s'_{x+i-1},s_{x+i}\}}$$

We start by exchanging the action s_{x+i-1} of I for the stronger action s'_{x+i-1} using the SHIFT principle. In general, SHIFT allows us to replace I with any interference assertion I' that has the same observable effect as far as the subjective assertion P is concerned (written $I \sqsubseteq^P I'$). In this instance, the actions s_{x+i-1} and s'_{x+i-1} have the same effect according to P, as discussed in the proof of slave(x). As such, rewriting s_{x+i-1} as s'_{x+i-1} merely reflects stronger knowledge about how $x{+}i$ and $x{+}i{-}2$ are related through $x{+}i{-}1$. In particular, $I \sqsubseteq^P (I \setminus \{s_{x+i-1}\}) \cup \{s'_{x+i-1}\}$ as required.

Next, because subjective views only describe *parts* of the shared state, we can use the FORGET principle to obtain a weaker view of the shared state, in this case a view that ignores all cells in the ring except for those at addresses $x{+}i{-}1$ and $x{+}i$. With all other cells out of the subjective view, their actions no longer have observable effects on the assertion, since they leave $x{+}i{-}1$ and $x{+}i$ unchanged. We can thus apply the SHIFT principle again to *frame off* those actions and obtain $S'_{x+i} = \{s'_{x+i-1}, s_{x+i}\}$.

Finally, we weaken the resulting subjective view so that it is stable with respect to S'_{x+i}, i.e. preserved by those of its actions that the environment may perform (here, s'_{x+i-1}). This yields the precondition of slave(x+i) as in Fig. 2. The preconditions of the master and foreman threads can be derived analogously.

Once all threads have completed their operations, we join up their post-conditions using the MERGE principle, which embodies a crucial feature of CoLoSL: different subjective views *overlap*. The overlapping conjunction $\curlywedge$ between two assertions means that the two assertions describe potentially *overlapping* pieces of state. In particular, $A * B \Rightarrow A \curlywedge B$ and $A \wedge B \Rightarrow A \curlywedge B$. This connective has been used in the past to reason about sharing in data structures [22,11,14]. Since $\vee$ distributes over $\curlywedge$, the subjective view simplifies to $\boxed{x \mapsto 10 * x{+}1 \mapsto 10 * \cdots * x{+}n \mapsto 10}_{I'}$ where $I' = M'_x \cup F'_{x+1} \cup S'_{x+2} \cup \cdots \cup S'_{x+n}$. Finally, since $I' \sqsubseteq^{\mathrm{n}\Rightarrow n * x\mapsto 10 * x+1\mapsto 10 * \cdots * x+n\mapsto 10} I$, we get the postcondition of ring(x) by the SHIFT principle. This concludes our CoLoSL proof of ring(x).

Small specifications and proof reuse. Our expansion and contraction of subjective views, in particular with the shifting of interference assertions at key places, enables us to confine the specification and verification of each thread to just the resources they need. Such small specifications make proofs robust against changes to each thread's environment, and provide more opportunities for proof reuse.

For instance, let us now add a thread that dynamically grows the ring by spawning extra slaves to the parallel composition of ring(x) (the details can be found in [20]). When the ring has $n{+}1$ machines, we use the EXTEND principle as follows

to add a new slave (at $x{+}n{+}1$) to the shared state with the associated interference relation and capability.[2]

$$(x{+}n{+}1) \mapsto v \overset{(\textsc{Extend})}{\Rrightarrow} \exists \mathsf{s}_{x+n+1}.\, [\mathsf{s}_{x+n+1}] * \boxed{x{+}n{+}1 \mapsto v}_{\{s_{x+n+1}\}}$$

Here, the *view shift* [6] (or *repartitioning* [7]) $P \Rrightarrow Q$ means that an instrumented (logical) state satisfying P may be changed to Q as long as the underlying machine state does not change. In particular, $(P \Rightarrow Q) \Rightarrow (P \Rrightarrow Q)$. The point is that our proof changes only minimally to accommodate the new program: the proof of master(x) accounts for new interference on $\mathrm{n} \Rrightarrow n$ since the environment can grow the ring, hence mutate n; the proofs of other threads can be directly reused.

In contrast, in existing CAP-like approaches [7,3], both $\mathrm{n} \Rrightarrow n$ and the global interference relation are observed by all threads. As such, with the above extension, the global interference relation needs to change (to include the interference on $\mathrm{n} \Rrightarrow n$), and the proofs of *both* master and slave threads need to be adapted.

3 CoLoSL

We now give a more formal overview of how to use CoLoSL for program verification, eschewing some details of the semantics for lack of space, while still providing enough ingredients to carry program proofs. We describe the underlying model of CoLoSL, give the semantics of CoLoSL assertions, then present proof rules to reduce the various obligations typically encountered in proofs (namely, the side-conditions of Shift and Extend, and stability checks) to classical separation logic entailments that, in particular, do not mention subjective views.

3.1 CoLoSL Worlds

A *world* is a triple $(l, g, \mathcal{I})$ where l and g are *logical states* and $\mathcal{I}$ is an *action model*. The *local logical state* (or local state) l represents the locally owned resources of a thread, in the standard separation logic sense, while the *global logical state* (or shared state) g represents shared resources. The action model $\mathcal{I}$ records all possible interferences on the shared state.

Logical states have two components: one describes machine states (e.g. stacks and heaps); the other represents *capabilities*. The latter are inspired by the capabilities in CAP [7]: a thread in possession of a given capability is allowed to perform the associated actions (as prescribed by the *action model* component of each world, defined below), while any capability *not* owned by a thread means that the environment can perform the action. This can be seen as a unified treatment of the rely and guarantee relations in rely-guarantee reasoning [16]: a capability

[2] Unlike CAP [7] and as in iCAP [23], we do not provide an explicit *unsharing* mechanism to claim back shared resources. Instead, this can be simply encoded as an action of the form $\bigstar_{j\in J}[\mathsf{a}_j]\colon P \rightsquigarrow \bigstar_{j\in J}[\mathsf{a}_j]$: a thread holding $\bigstar_{j\in J}[\mathsf{a}_j]$ can move the shared resource P into its local state in exchange for the associated capabilities.

fully owned (resp. fully not owned) during the entire lifetime of a thread represents its guarantee (resp. its rely), while a partially-owned capability means that the corresponding action is both in the rely and the guarantee. Capabilities go beyond the rely-guarantee model [9]; in particular, they may be transferred between a thread and its environment just like any other resource to temporarily block or enable certain actions. See the presentation of CAP [7] and deny-guarantee [9] for further details and motivation.

In general, each component of a logical state is taken from an arbitrary *separation algebra* [2] (i.e. a cancellative, partial commutative monoid) that satisfies the *cross-split* property[3][8] (this is needed for $\uplus$ to be associative [14]). As we demonstrate in the examples of §4, our programs often call for a more complex model of machine states and capabilities than that presented here. For instance, we may need our capabilities to be fractionally owned, where ownership of a *fraction* of a capability grants the right to perform the action to both the thread and the environment, while a fully-owned capability by the thread *denies* the right to the environment to perform the associated action. For ease of presentation, the focus of this paper is on the standard stack and heap model for machine states, and finite sets of *tokens* (which are simple names) for capabilities. We assume a set of program values Val, as well as infinite disjoint sets PVar, Loc, and Token of program variables, memory locations, and tokens, respectively.

Definition 1 (Logical states). *A* logical state *is a tuple* $((\sigma, h), \kappa) \in \mathsf{LState}$*, also written* (σ, h, κ)*, of a finite partial* stack $\sigma \in \mathsf{Stack}$ *associating program variables with values, a* heap $h \in \mathsf{Heap}$ *associating heap locations with values, and a* capability $\kappa \in \mathbb{K}$*:*

$$\mathsf{Stack} \triangleq \mathsf{PVar} \rightharpoonup_{fin} \mathsf{Val} \qquad \mathsf{Heap} \triangleq \mathsf{Loc} \rightharpoonup_{fin} \mathsf{Val} \qquad \mathbb{M} \triangleq \mathsf{Stack} \times \mathsf{Heap}$$

$$\mathbb{K} \triangleq \mathcal{P}_{fin}(\mathsf{Token}) \qquad \mathsf{LState} \triangleq \mathbb{M} \times \mathbb{K}$$

The local and global logical states of a world are always *compatible*: they can be composed with one another. This captures the intuition that locally-owned resources are disjoint from shared resources. The composition of logical states is defined component-wise as disjoint function union $\uplus$ over stacks and heaps, and disjoint set union $\uplus$ on capabilities.

Definition 2 (Logical state composition). *The* composition of logical states $\circ : \mathsf{LState} \times \mathsf{LState} \rightharpoonup \mathsf{LState}$ *is defined as:*

$$((\sigma, h), \kappa) \circ ((\sigma', h'), \kappa') \triangleq ((\sigma \uplus \sigma', h \uplus h'), \kappa \uplus \kappa')$$

We write l to range over arbitrary logical states or just local states, and g to range over logical states representing global shared states. The empty logical state $(\emptyset, \emptyset, \emptyset)$ is written $\mathbf{0}_{\mathsf{L}}$. We write $l_1 \leq l_2$ when there exists l such that $l \circ l_1 = l_2$, and

[3] A monoid $(\mathbb{A}, +, \mathbf{1})$ satisfies the cross-split property iff, for all $a, b, c, d \in \mathbb{A}$, if $a + b = c + d$, there exists $x, y, z, w \in \mathbb{A}$ s.t. $a = x + y$, $b = z + w$, $c = x + z$ and $d = y + w$.

write $l_2 - l_1$ to denote the unique such l (by cancellativity) when it exists. When $l_1 \circ l_2$ is defined, we say that l_1 and l_2 are *compatible* and write $l_1 \sharp l_2$.

An action is a triple (p, q, c) of logical states where p and q are the action *pre-* and *post-states* describing how the shared state is modified by the action; and c is the action *catalyst*. An action catalyst has to be present for the action to take effect, but is left unchanged by the action. It is maximal in the sense that no further, non-empty catalyst c' can be found, which we write $p \bot q$: $\forall l.\, l \leq p \wedge l \leq q \implies l = \mathbf{0}_{\mathsf{L}}$.[4] For instance, as we shall shortly see, s_x in Fig. 2 will be interpreted as the set of actions $S = \{a_v \triangleq ((\emptyset, \{\ell : v\}, \emptyset), (\emptyset, \{\ell : v{+}1\}, \emptyset), (\emptyset, \{\ell{-}1 : v{+}1\}, \emptyset)) \mid v \in \mathbb{N}\}$, where ℓ is the value of x in the current logical environment.

An action model is a partial function from *capabilities* to sets of *actions*. It corresponds to the semantic interpretation of an interference assertion.

Definition 3 (Action models). *The set* Action *of* actions*, ranged over by* a*, and the set* AMod *of* action models*, ranged over by* $\mathcal{I}$*, are defined as:*

$$\mathsf{Action} \triangleq \mathsf{LState} \times \mathsf{LState} \times \mathsf{LState} \qquad \mathsf{AMod} \triangleq \mathbb{K} \rightharpoonup \mathcal{P}(\mathsf{Action})$$

Worlds are triples $(l, g, \mathcal{I}) \in \mathsf{LState} \times \mathsf{LState} \times \mathsf{AMod}$ that satisfy several well-formedness conditions: the local and shared states are compatible; the capabilities owned by l and g are in the domain of $\mathcal{I}$; and actions in $\mathcal{I}$ are *confined* to g (written $g \circledcirc \mathcal{I}$). An action $a = (p, q, c)$ is confined to g if and only if, whenever it is enabled ($p \circ c$ agrees with g), then its pre-state p is contained in g ($p \leq g$). We motivate the need for the confinement condition in §3.3.

Definition 4 (Well-formedness). *A triple* $(l, g, \mathcal{I}) \in \mathsf{LState} \times \mathsf{LState} \times \mathsf{AMod}$ *is* well-formed*, written* $\mathsf{wf}\,(l, g, \mathcal{I})$*, iff* $l \sharp g$*,* $l_\kappa \cup g_\kappa \subseteq \bigcup dom(\mathcal{I})$ *and* $g \circledcirc \mathcal{I}$*.*

Definition 5 (Worlds). *The set* World *of* worlds *consists of all well-formed triples:*

$$\mathsf{World} \triangleq \{w \in \mathsf{LState} \times \mathsf{LState} \times \mathsf{AMod} \mid \mathsf{wf}\,(w)\}$$

Finally, the composition of two worlds is defined when their local states are compatible, their global shared states and action models are the same, and the resulting tuple is well-formed.

Definition 6 (World composition). *The* composition *of worlds,* $\bullet : \mathsf{World} \times \mathsf{World} \rightharpoonup \mathsf{World}$*, is defined as:*

$$(l, g, \mathcal{I}) \bullet (l', g', \mathcal{I}') \triangleq \begin{cases} (l \circ l', g, \mathcal{I}) & \text{if } g = g',\ \text{and } \mathcal{I} = \mathcal{I}',\ \text{and } \mathsf{wf}((l \circ l', g, \mathcal{I})) \\ \text{undefined} & \text{otherwise} \end{cases}$$

[4] Alternatively, the catalyst could be computed a posteriori for each action. However, we often need to isolate the part of the state that is modified by an action, hence our technical choice of recording the catalyst in the model.

3.2 CoLoSL Assertions

Our assertion language extends separation logic with *subjective views* and *capability assertions*.

CoLoSL is parametric in the assertions of machine states and capabilities, and can be instantiated with any assertion language over machine states $\mathbb{M}$ and capabilities $\mathbb{K}$. In this paper, we use standard heap and stack assertions for machine state assertions, and single token assertions of the form $[\mathsf{a}]$ for capability assertions where $\mathsf{a} \in \mathsf{Token}$. We write $[\mathsf{A}]$ as a shorthand for $\bigstar_{\mathsf{a}\in\mathsf{A}}[\mathsf{a}]$. We assume an infinite set LVar of *logical variables*, disjoint from PVar.

Definition 7 (Assertion syntax). *Given* $x \in \mathsf{LVar}$, $\mathrm{x} \in \mathsf{PVar}$, *and* $\mathsf{a} \in \mathsf{Token}$, the assertions of CoLoSL , Assn, *are described by the grammars below:*

$$\begin{aligned}
\mathsf{LAssn} \ni p, q ::= {} & \mathit{false} \mid E_1 = E_2 \mid \mathsf{emp} \mid \mathrm{x} \Rightarrow E \mid E_1 \mapsto E_2 \mid [E] \\
& \mid p \vee q \mid \neg p \mid \exists x.\, p \mid p \ast q \mid p \uplus q \mid p \mathrel{-\!\!\circledast} q \\
\mathsf{Assn} \ni P, Q ::= {} & p \mid \exists x.\, P \mid P \vee Q \mid P \ast Q \mid P \uplus Q \mid \boxed{P}_I \\
I ::= {} & \emptyset \mid \{[\mathsf{A}]\colon \exists \bar{x}.\, P \rightsquigarrow Q\} \cup I \qquad E ::= x \mid \mathsf{a} \mid E_1 + E_2 \mid \cdots
\end{aligned}$$

The syntax and semantics of *local assertions* LAssn are as in standard separation logic with variables-as-resource [1].[5] Local assertions are interpreted over a world's local state. The empty local state $\mathbf{0}_{\mathsf{L}}$ is denoted by emp. The assertion $\mathrm{x} \Rightarrow E$ denotes a singleton stack where x has value E. Similarly, $E_1 \mapsto E_2$ is true of the singleton heap where only address E_1 is allocated and has value E_2. The capability assertion $[E]$ is true of the singleton capability $\{\mathsf{a}\}$ if E evaluates to a. The *separating conjunction* $p \ast q$ is true when the local state can be split into two according to $\circ$ such that one state satisfies p and the other satisfies q. The *overlapping conjunction* $p \uplus q$ is true when the local state can be split three-ways according to $\circ$, such that the $\circ$-composition of the first two states satisfies p and the $\circ$-composition of the last two satisfy q [14,12,22]. *Septraction* (or *existential magic wand*) $p \mathrel{-\!\!\circledast} q$ is true when there exists a local state satisfying p that can be $\circ$-composed with the current one to yield a state satisfying q. The usual predicates and connectives have their standard classical meaning.

As in RGSep [26], our assertions Assn are defined on top of local assertions. For simplicity, assertions do not include negation nor septraction. The interpretation of assertions is a simple lift from that of local assertions, with the exception of the subjective view $\boxed{P}_I$. First, an *interference assertion* I describes actions enabled by a given capability, in the form of a pre- and postcondition. A subjective view $\boxed{P}_I$ is then true of $(l, g, \mathcal{I})$ when $l = \mathbf{0}_{\mathsf{L}}$ and a *subjective state* s can be found in the global shared state g (i.e. $g = s \circ r$ for some r), such that s satisfies P, and $\mathcal{I}$ and I *agree* given the decomposition s and r, written $\mathcal{I} \downarrow (s, r, [\![I]\!]_\iota)$, in the following sense:

[5] Note in particular that expressions E do not allow program variables: they can only appear on the left-hand side of $\mathrm{x} \Rightarrow E$.

1. every action in I is reflected in $\mathcal{I}$;
2. every action in $\mathcal{I}$ that has a visible effect on s is reflected in I;
3. the above holds after any number of actions in $\mathcal{I}$ takes place.

Thus, given a world $(l, g, \mathcal{I})$ and a subjective view $\boxed{P}_I$, P describes a subjective state s that is *a part* of g and I describes *all parts* of $\mathcal{I}$ concerning s, while $\mathcal{I}$ describes the overall interference on g. We refer to the above agreement between the action model and the subjective view as the *action model closure property*. We omit its formal definition for lack of space.

The semantics of CoLoSL assertions is given by a forcing relation $w, \iota \vDash P$ between a world w, a logical environment $\iota \in \mathsf{LEnv}$, and an assertion P. We use two auxiliary forcing relations. The first one $l, \iota \vDash_{\mathsf{SL}} p$ interprets local assertions $p \in \mathsf{LAssn}$ in the usual separation logic sense over a logical state l. The second one $s, \iota \vDash_{g,\mathcal{I}} P$ interprets assertions $P \in \mathsf{Assn}$ over a *subjective state* s that is part of the global shared state g, subject to the action model $\mathcal{I}$. This second relation is needed to deal with the nesting of subjective views.[6] Since logical connectives are interpreted uniformly in all cases, we write $\vDash_\dagger$ for any of the three satisfaction relations, u for elements of either World or LState, and $\bullet$ for either $\bullet$ or $\circ$, as appropriate.

Definition 8 (Assertion semantics). *Given a logical environment* $\iota : \mathsf{LVar} \to \mathsf{Val}$*, the semantics of CoLoSL assertions is defined below, using the semantics of interference assertions* $[\![.]\!]_{(.)} : \mathsf{LEnv} \to \mathsf{AMod}$ *also defined below:*

$$
\begin{array}{lll}
(l, g, \mathcal{I}), \iota \vDash p & \textit{iff } l, \iota \vDash_{\mathsf{SL}} p & \\
(l, g, \mathcal{I}), \iota \vDash \boxed{P}_I & \textit{iff } l = \mathbf{0}_{\mathsf{L}} \textit{ and } \exists s, r.\, g = s \circ r \textit{ and } s, \iota \vDash_{g,\mathcal{I}} P \textit{ and } \mathcal{I}{\downarrow}\,(s, r, [\![I]\!]_\iota) & \\
s, \iota \vDash_{g,\mathcal{I}} p & \textit{iff } s, \iota \vDash_{\mathsf{SL}} p & \\
s, \iota \vDash_{g,\mathcal{I}} \boxed{P}_I & \textit{iff } (s, g, \mathcal{I}), \iota \vDash \boxed{P}_I & \\
u, \iota \vDash_\dagger \exists x.\, P & \textit{iff } \exists v.\, u, [\iota \mid x : v] \vDash_\dagger P & \\
u, \iota \vDash_\dagger P \vee Q & \textit{iff } u, \iota \vDash_\dagger P \textit{ or } u, \iota \vDash_\dagger Q & \\
u, \iota \vDash_\dagger P_1 * P_2 & \textit{iff } \exists u_1, u_2.\, u = u_1 \bullet u_2 \textit{ and } u_1, \iota \vDash_\dagger P_1 \textit{ and } u_2, \iota \vDash_\dagger P_2 & \\
u, \iota \vDash_\dagger P_1 \circledast P_2 & \textit{iff } \exists u', u_1, u_2.\, u = u' \bullet u_1 \bullet u_2 \textit{ and} & \\
 & \quad u' \bullet u_1, \iota \vDash_\dagger P_1 \textit{ and } u' \bullet u_2, \iota \vDash_\dagger P_2 & \\
l, \iota \vDash_{\mathsf{SL}} \mathit{false} & \quad \textit{never} & \\
l, \iota \vDash_{\mathsf{SL}} \mathsf{emp} & \textit{iff } l = \mathbf{0}_{\mathsf{L}} & \\
l, \iota \vDash_{\mathsf{SL}} E_1 = E_2 & \textit{iff } [\![E_1]\!]_\iota = [\![E_2]\!]_\iota & \\
l, \iota \vDash_{\mathsf{SL}} \mathrm{x} \Rrightarrow E & \textit{iff } l = (\{\mathrm{x} : [\![E]\!]_\iota\}, \emptyset, \emptyset) & \\
l, \iota \vDash_{\mathsf{SL}} E_1 \mapsto E_2 & \textit{iff } l = (\emptyset, \{[\![E_1]\!]_\iota : [\![E_2]\!]_\iota\}, \emptyset) & \\
l, \iota \vDash_{\mathsf{SL}} [E] & \textit{iff } l = (\emptyset, \emptyset, \{[\![E]\!]_\iota\}) & \\
l, \iota \vDash_{\mathsf{SL}} \neg p & \textit{iff } l, \iota \nvDash_{\mathsf{SL}} p & \\
l, \iota \vDash_{\mathsf{SL}} p \mathrel{-\!\!\circledast} q & \textit{iff } \exists l'.\, l', \iota \vDash_{\mathsf{SL}} p \textit{ and } l \sharp l' \textit{ and } l \circ l', \iota \vDash_{\mathsf{SL}} q &
\end{array}
$$

[6] This presentation with several forcing relations differs from the usual CAP presentation [7], where assertions are first interpreted over worlds that are not necessarily well-formed, and then cut down to well-formed ones. The advantage of our presentation is that the semantics of assertions is *compositional*, e.g. the semantics of $P * Q$ follows directly from the semantics of P and Q.

$$[\![I]\!]_\iota(\mathsf{A}) \triangleq \left\{ (p,q,c) \;\middle|\; \begin{array}{l} [\mathsf{A}]\colon \exists \bar{x}.\, P \rightsquigarrow Q \in I \wedge p \perp q \wedge \exists \bar{v}, \mathcal{I}, r, r'.\, c = r \circ r' \wedge \\ \quad p \circ r, [\iota \mid \bar{x} : \bar{v}] \vDash_{p \circledcirc c, \mathcal{I}} P \wedge q \circ r, [\iota \mid \bar{x} : \bar{v}] \vDash_{q \circledcirc c, \mathcal{I}} Q \end{array} \right\}$$

Note that, as in the CAP family [7,23,3], CoLoSL cannot ensure that proved programs do not leak memory. This is because of the following property of the semantics with respect to the shared state (sometimes called "intuitionistic semantics" [21]): if $(l, g, \mathcal{I}), \iota \vDash P$ then $(l, g \circ g', \mathcal{I}), \iota \vDash P$.

Five of the principles of Fig. 1 are direct consequences of the semantics.

Lemma 1. *The CoLoSL reasoning principles* Forget, Merge, Shift, Weaken, *and* Copy *are valid.*

Proof (sketch). The cases of Weaken and Copy are straightforward. For Forget, Merge, and Shift, we note in [20] that action model closure is preserved by picking a smaller subjective state, taking the union of subjective states and their interference assertions, and shifting the interference assertion, respectively. □

3.3 Reducing CoLoSL Principles to Separation Logic Entailments

We turn to the remaining two principles, Extend and Shift, and to the stability of assertions. These involve reasoning outside our assertion language, potentially requiring semantic reasoning in the model. Fortunately, it is enough to work with a partial axiomatisation for all three conditions to verify our examples. In this section, we give cut-down versions of these rules for a fragment of the CoLoSL assertion language where the nesting of subjective views is not permitted and interference assertions cannot mention subjective views. This restriction is easily lifted: assertions with nested boxes can always be *flatten* into logically equivalent assertions with no nesting; and interference assertions mentioning other subjective views in their actions may be rewritten into ones that do not. See [20] for the full details.

Confinement. The soundness of CoLoSL hinges on the fact that, given a world $(l, g, \mathcal{I})$, the action model $\mathcal{I}$ contains all actions that could possibly affect the shared state g. This was captured by a well-formedness condition in the definition of worlds (Def. 5) as $g \circledcirc \mathcal{I}$, stipulating that the actions in $\mathcal{I}$ are confined to the shared state g. It is also possible to extend g at any time. Any part l' of the local state can migrate to the shared state under a new set of actions $\mathcal{I}'$, yielding a new shared state $g \circ l'$ and action model $\mathcal{I} \cup \mathcal{I}'$. This migration is only permitted if $l' \circledcirc \mathcal{I}'$. This confinement condition means that the extension does not invalidate the views of the threads.

The technical definition of confinement of an action model on a logical state is given in the technical report [20]. Intuitively, it means that, whenever an action $a = (p, q, c)$ of $\mathcal{I}'$ is enabled, the pre-state p must be a substate of l'. It is possible for some of the catalyst c to lie outside l', since the fact that it does not change during the course of the action means that it will not have an effect on the views of other threads. For example, recall the interpretation of s_x given by S just before Def. 3. The action $a_0 = ((\emptyset, \{\ell : 0\}, \emptyset), (\emptyset, \{\ell : 1\}, \emptyset), (\emptyset, \{\ell{-}1 : 1\}, \emptyset)) \in S$ is confined to

$$\frac{P \vdash f \quad f \blacktriangleright I}{P \mathbin{©} I} \qquad \frac{\forall\, ([\mathsf{A}]\colon \exists \bar{x}.\, p \rightsquigarrow q) \in I.\, f \blacktriangleright \{[\mathsf{A}]\colon \exists \bar{x}.\, p \rightsquigarrow q\}}{f \blacktriangleright I} \qquad \frac{f \circledast p \vdash_{\mathsf{SL}} \mathit{false}}{f \blacktriangleright \{[\mathsf{A}]\colon \exists \bar{x}.\, p \rightsquigarrow q\}}$$

$$\frac{p \vdash_{\mathsf{SL}} p' * r \qquad q \vdash_{\mathsf{SL}} q' * r \qquad \mathsf{exact}(r) \qquad f \blacktriangleright \{[\mathsf{A}]\colon \exists \bar{x}.\, p' \rightsquigarrow q'\}}{f \blacktriangleright \{[\mathsf{A}]\colon \exists \bar{x}.\, p \rightsquigarrow q\}}$$

$$\frac{(p \mathbin{-\!\!\circledast} f) * q \vdash_{\mathsf{SL}} f \qquad f \Leftrightarrow \bigvee_{i \in J} f_i \qquad (\mathsf{precise}\,(f_i) \wedge f_i \circledast p \vdash_{\mathsf{SL}} f_i \ \text{ for } i \in J)}{f \blacktriangleright \{[\mathsf{A}]\colon \exists \bar{x}.\, p \rightsquigarrow q\}}$$

Fig. 3. Selected confinement and local fencing rules. We assume that variables in $\bar{x}$ do not appear free in f, and in f_i for any i.

the logical state $l_0 = (\emptyset, \{\ell : 0\}, \emptyset)$ because l_0 is the first component of the action, and state $(\emptyset, \{\ell{-}1 : 0\}, \emptyset)$ because it is incompatible with the action, but not state $(\emptyset, \{\ell{-}1 : 1\}, \emptyset)$ because the action can potentially affect address ℓ. The definition of confinement also requires that all states resulting from the successive application of actions in $\mathcal{I}'$ themselves confine all actions in $\mathcal{I}'$. For instance, we also require that $l_1 = (\emptyset, \{\ell : 1\}, \emptyset)$ resulting from the application of a_0 on l_0, $l_2 = (\emptyset, \{\ell : 2\}, \emptyset)$ resulting from the application of a_1 on l_1, and so forth, *all* confine the actions in S. Inspired by the LRG approach [10], we achieve this by first finding a set that is invariant under all actions in $\mathcal{I}'$ (called a *fence*), then checking the confinement condition for each action. We provide the technical details of confinement in [20].

For our examples, it is in fact enough to work with confinement in the logic. We lift the notion of confinement to assertions, and write $P \mathbin{©} I$ when the set of states described by P confines the actions of interface assertion I. In the logic, the shared state can be extended by the resources in P under the interference assertion I via the Extend principle, which requires that $P \mathbin{©} I$ be established.

Fig. 3 presents a set of rules which reduces $P \mathbin{©} I$ to a series of entailments in ordinary separation logic. As expressed by the first rule, $P \mathbin{©} I$ holds if there is a weaker local assertion f that acts as a *local fence* for I, written $f \blacktriangleright I$. This relation states that f must be invariant under all actions of I and must confine the actions in f. In [20], we show that it is always possible to weaken an arbitrary assertion into a local assertion. This fencing condition is checked for each action in I (see the second rule). For each action $[\mathsf{A}]\colon \exists \bar{x}.\, p \rightsquigarrow q$, the three remaining rules of the figure may apply. In the first of these rules, the action cannot possibly fire, because its precondition does not agree with f: no state satisfying f may be extended such that a subpart satisfies p. The second of these rules allows us to trim a *neutral* part r (corresponding to a part of the catalyst in the interpretation of this action) of an action $[\mathsf{A}]\colon \exists \bar{x}.\, p \rightsquigarrow q$ appearing both in p and q. This only applies when r is *exact*, i.e. satisfied by at most one logical state:[7] the part of the state denoted by r is then uniquely determined and left unchanged by the action. The last rule finally reduces local fencing to entailment checking, provided the fence f can be expressed as a disjunction of *precise* assertions, i.e. assertions satisfied

[7] $\mathsf{exact}(p) \triangleq \forall \iota, l_1, l_2.\, l_1, \iota \vDash_{\mathsf{SL}} p$ and $l_2, \iota \vDash_{\mathsf{SL}} p$ implies $l_1 = l_2$

$$\frac{P \vdash f \qquad I \sqsubseteq^f I'}{I \sqsubseteq^P I'} \qquad \frac{}{I \sqsubseteq^f I} \qquad \frac{f \rhd I \cup I_1 \qquad I_1 \sqsubseteq^f I_2}{I \cup I_1 \sqsubseteq^f I \cup I_2}$$

$$\frac{p \vdash_{\mathsf{SL}} p' * r \qquad q \vdash_{\mathsf{SL}} q' * r \qquad \mathsf{exact}(r) \qquad f \perp p'}{\{[\mathsf{A}]\colon \exists \bar{x}.\, p \rightsquigarrow q\} \sqsubseteq^f \emptyset} \qquad \frac{(p \mathrel{-\!\!\circledast} (f \uplus p)) * q \vdash_{\mathsf{SL}} \mathit{false}}{\{[\mathsf{A}] : \exists \bar{x}.\, p \rightsquigarrow q\} \sqsubseteq^f \emptyset}$$

$$\frac{f \uplus p \vdash_{\mathsf{SL}} \bigvee_{i \in J} f \uplus (p * r_i) \qquad \mathsf{exact}(r_i) \text{ for } i \in J}{\{[\mathsf{A}]\colon \exists \bar{x}.\, p \rightsquigarrow q\} \;\equiv^f\; \bigcup_{i \in J} \{[\mathsf{A}]\colon \exists \bar{x}.\, p * r_i \rightsquigarrow q * r_i\}}$$

$$\frac{}{\bigcup_{i \in K, j \in J} \{[\mathsf{A}] : \exists \bar{x}.\, p_i \rightsquigarrow q_j\} \equiv^{\mathit{true}} \left\{[\mathsf{A}] : \exists \bar{x}.\, \bigvee_{i \in K} p_i \rightsquigarrow \bigvee_{j \in J} q_j\right\}} \qquad \frac{}{\mathit{true} \rhd I}$$

$$\frac{\forall [\mathsf{A}] : \exists \bar{x}.\, p \rightsquigarrow q \in I.\, f \rhd \{[\mathsf{A}] : \exists \bar{x}.\, p \rightsquigarrow q\}}{f \rhd I} \qquad \frac{p \perp q \qquad (p \mathrel{-\!\!\circledast} (f \uplus p)) * q \vdash_{\mathsf{SL}} f}{f \rhd \{[\mathsf{A}]\colon \exists \bar{x}.\, p \rightsquigarrow q\}}$$

$$\frac{f \perp p}{f \rhd \{[\mathsf{A}]\colon \exists \bar{x}.\, p \rightsquigarrow q\}} \qquad \frac{p \vdash_{\mathsf{SL}} p' * r \quad q \vdash_{\mathsf{SL}} q' * r \quad \mathsf{exact}(r) \quad f \rhd \{[\mathsf{A}]\colon \exists \bar{x}.\, p' \rightsquigarrow q'\}}{f \rhd \{[\mathsf{A}]\colon \exists \bar{x}.\, p \rightsquigarrow q\}}$$

Fig. 4. Selected action shifting rules. We write $I \equiv^f I'$ for $I \sqsubseteq^f I' \wedge I' \sqsubseteq^f I$, and $p \perp q$ to denote that states satisfying p and q have empty intersections. We assume that variables in $\bar{x}$ do not appear free in f.

by at most one substate of each logical state.[8] The first premise states that f is invariant under the action $[\mathsf{A}]\colon \exists \bar{x}.\, p \rightsquigarrow q$, similar to the way that RGSep encodes stability checks as separation logic entailments. Informally, it reads: for any state in f, remove a part satisfying p, add a state satisfying q, and the result should still be in f. The third premise checks the confinement condition: given a state in local assertion f_i ($l_1 \circ l_2 \vDash f_i$), and a state in p ($l_2 \circ l_3 \vDash p$ with l_1 and l_3 disjoint), the combined state ($l_1 \circ l_2 \circ l_3$) must also be in f_i. Hence, by precision of f_i, we have $l_1 \circ l_2 \circ l_3 = l_1 \circ l_2$, i.e. $l_2 \circ l_3 \leq l_1 \circ l_2$ as required.

Shifting and fencing. Fig. 4 presents a partial axiomatisation of the shifting condition $I \sqsubseteq^P I'$ required by the SHIFT principle. As with confinement, we omit the direct semantic definition. Intuitively, the relation $I \sqsubseteq^P I'$ means that interference assertion I can be replaced by I', because I and I' describe the same interference with respect to the states described by P. The first rule weakens the shifting condition from assertions to local fence assertions. The third rule reduces the shifting judgement $I \cup I_1 \sqsubseteq^f I \cup I_2$ to the simpler $I_1 \sqsubseteq^f I_2$, provided that f is invariant with respect to $I \cup I_1$, written $f \rhd I \cup I_1$. This invariant fencing condition is necessary: $I_1 \sqsubseteq^f I_2$ only states that I_1 and I_2 have the same effect with respect to f; $f \rhd I \cup I_1$ states that f is an *invariant* of the shared state under the combined interferences of I and I_1.

The next two rules describe situations where it is impossible to apply the action to f: when the precondition of the action is entirely outside f; or when the

[8] $\mathsf{precise}\,(p) \triangleq \forall \iota, l, l_1, l_2.\, l_1 \leq l,\, l_2 \leq l,\, l_1, \iota \vDash_{\mathsf{SL}} p$, and $l_2, \iota \vDash_{\mathsf{SL}} p$ implies $l_1 = l_2$

postcondition is incompatible with that part of f not associated with the precondition. The notation $p \perp q$ asserts that states described by p and q have *empty intersections*: whenever $l_1, \iota \vDash_{\mathsf{SL}} p$ and $l_2, \iota \vDash_{\mathsf{SL}} q$, we have $l_1 \perp l_2$ as defined in §3.1. This is expressible in standard separation logic as follows:

$$p \perp q \Leftrightarrow p \vdash_{\mathsf{SL}} \neg\,(\mathit{true} * (\neg\mathsf{emp} \wedge (\mathit{true} \mathrel{-\!\!\circledast} q)))$$

The next rule is a shifting equivalence that uses the knowledge embodied by f of all the possible states that the subjective shared state may be in to rewrite an action into an equivalent one. More precisely, if whenever the precondition p of the action agrees with f then one of the r_i's is also true, then adding r_i as a neutral part of the action produces the same behaviour. We can use this rule (with the single $r_0 = x \mapsto v$) to justify the shiftings of §2. The fact that the r_i's are exact guarantees that no piece of state in r_i is mutated by the "larger" action. In general, it may not be the case that a single exact assertion can be added, but it may be the case that a disjunction of exact facts holds. The last shifting equivalence is straightforward.

The last five rules partially axiomatise the *fencing* relation $f \rhd I$. Most are similar to those for local fencing $f \blacktriangleright I$. The first two state that *true* fences any interference assertion, and fencing can be checked per action. The last one states that, as for local fencing, neutral parts of actions may be ignored. The one before that states that, contrarily to local fencing, actions are allowed to have effects outside of the fence. If the action precondition does not intersect with the fence, then its effect is entirely outside the fence and the action may be ignored.

Let us now focus on the third of these rules, which states that whenever p and q do not intersect in an action $[\mathsf{A}]\colon \exists \bar{x}.\, p \rightsquigarrow q$ (e.g. when their common parts have been removed using the last rule), then the application of the action must preserve the fence f. Contrarily to the case of local fencing, the action is allowed to act partly outside of f, hence the state on which the action is applied is $f \uplus p$. However, the whole of the postcondition q of the action is then added, and the resulting state must still be in f. One might instead have expected that only parts of the resulting state need to be represented in f, to mimic the relationship between f and p (and indeed, this is all that is required for stability, as we shall see next). However, we do need the full q. Recall that shifting asserts that two interference assertions have the same effect even after an arbitrary number of steps. Doing otherwise would be unsound because there would be no guarantee that I accounts for all possible actions on that part of q that would be discarded, since it would not be part of f. Hence, we could end up with a new interference assertion I' that breaks the original action model closure property. Finally, the reason why p and q must not intersect stems from similar considerations. If p and q had a non-empty intersection c, such that c is not part of f, then this would force the fence to account for c which would prevent us from forgetting actions associated with it using shifting. For instance, it would make it impossible to forget actions as we do in §2.

Stability. Fig. 5 partially axiomatises the stability of assertions of the form $\exists \bar{x}.\, p * \bigstar_{i \in J} \boxed{q_i}_{I_i}$. We work with a more restricted form of assertions than our two previous axiomatisations for simplicity; see [20] for the general case. The first two

$$\frac{}{\mathsf{Stable}(p)} \qquad \frac{\mathsf{Stable}(P)}{\mathsf{Stable}(\exists x.\, P)} \qquad \frac{\forall j \in J.\, \mathsf{Stable}(q_j, I_j, p, \textcircled{\ast}_{i\in J}\, q_i)}{\mathsf{Stable}(p \ast \mathbin{\textcircled{\ast}}_{i\in J} \boxed{q_i}_{I_i})}$$

$$\frac{\forall [\mathsf{A}]\colon \exists \bar{x}.\, p_1 \rightsquigarrow p_2 \in I.\, \mathsf{Stable}(q, \{[\mathsf{A}]\colon \exists \bar{x}.\, p_1 \rightsquigarrow p_2\}, p, r)}{\mathsf{Stable}(q, I, p, r)} \qquad \frac{[\mathsf{A}] \ast p \ast r \vdash_{\mathsf{SL}} \mathit{false}}{\mathsf{Stable}(q, \{[\mathsf{A}]\colon \exists \bar{x}.\, p_1 \rightsquigarrow p_2\}, p, r)}$$

$$\frac{p \ast (r \textcircled{\ast} p_1) \vdash_{\mathsf{SL}} \mathit{false}}{\mathsf{Stable}(q, \{[\mathsf{A}]\colon \exists \bar{x}.\, p_1 \rightsquigarrow p_2\}, p, r)} \qquad \frac{(p_1 \mathbin{-\!\!\textcircled{\ast}} (q \textcircled{\ast} p_1)) \ast p_2 \vdash_{\mathsf{SL}} q \ast \mathit{true}}{\mathsf{Stable}(q, \{[\mathsf{A}]\colon \exists \bar{x}.\, p_1 \rightsquigarrow p_2\}, p, r)}$$

Fig. 5. Selected rules for stability checks. We assume that variables in $\bar{x}$ do not appear free in p, q, and r.

rules state that local assertions are always stable and existentials can be eliminated. The next rule states that checking the stability of $p \ast \mathbin{\textcircled{\ast}}_{i\in J} \boxed{q_i}_{I_i}$ boils down to establishing, for each local assertion q_j for $j \in J$, that the four-place predicate $\mathsf{Stable}(q_j, I_j, p, \textcircled{\ast}_{i\in J}\, q_i)$ holds: this means that q_i is stable under the interference assertion I_i, a local context p, and a shared context made of the $\textcircled{\ast}$-combination of all the subjective assertions (including q_j). In turn, checking this fact reduces to checking stability for each action of I_j. The last three rules deal with checking stability for a single action, in a similar way to the fencing rules above.

The first of these rules is unfamiliar. Unlike fencing, stability checking for assertions only has to be checked against actions for which the environment may have the capability. If the capability required by the action cannot exist separately from those held by the assertion (that is, those in $p \ast r$) then the environment cannot possibly own the capability to perform the action. Similarly, an action whose precondition is incompatible with the assertion $p \ast r$ cannot possibly fire, as stated by the next rule. The last rule checks that q is preserved by the effect of an action. Again, there is a crucial difference with the corresponding check for fencing: in p_2, the action may bring in some newly-shared state, hence the result $q \ast \mathit{true}$; but the FORGET rule allows us to immediately discard it, if appropriate.

Proof reuse. Note that, once the local fencing judgement $f \blacktriangleright I$ has been established, it automatically establishes the weaker fencing judgement $f \rhd I$, which in turn implies that f is stable under I. Also, fencing is preserved by action shifting. These observations provide the admissible rules:

$$\frac{f \blacktriangleright I}{f \rhd I} \qquad \frac{p \rhd I}{\mathsf{Stable}(\boxed{p}_I)} \qquad \frac{f \rhd I \qquad I \sqsubseteq^f I'}{f \rhd I'}$$

The other directions are not valid in general: fencing lacks the confinement condition required by local fencing; stability of p under I may omit part of the state resulting from an action application to re-establish p, which is not allowed in fencing; and a fence f for a smaller interference assertion need not be a fence for a larger interference assertion.

3.4 Soundness

We prove the soundness of CoLoSL, parametrised by the underlying models of machine states ($\mathbb{M}$) and capabilities ($\mathbb{K}$). We appeal to the general soundness result of the views framework, providing parameters such as the reification function in Def. 9 and proving lemmas required to make the result hold.

The main part of the proof establishes the soundness of the following rule for atomic commands, where $\vdash_{\mathsf{SL}}$ represents standard sequential separation logic:

$$\frac{\vdash_{\mathsf{SL}} \{p\}\ \mathbb{C}\ \{q\} \quad P \Rrightarrow^{\{p\}\{q\}} Q}{\vdash\ \{P\}\ \langle \mathbb{C} \rangle\ \{Q\}}\ \textsc{Atom}$$

This rule is present in logics arising from CAP [7,23]. For CoLoSL, the *repartitioning* $P \Rrightarrow^{\{p\}\{q\}} Q$ holds if, from any world $(l, g, \mathcal{I})$ satisfying P, whenever parts of $l \circ g$ satisfies p then substituting that part for any other satisfying q will yield a state $l' \circ g'$ such that there exists $\mathcal{I}'$ such that $(l', g', \mathcal{I}')$ satisfies Q. Moreover, the passage from $(l, g, \mathcal{I})$ to $(l', g', \mathcal{I}')$ can be achieved via a succession of valid updates from $\mathcal{I}$ and valid extension steps. The details can be found in the accompanying technical report [20].

Semantic validity of Hoare triples depends on the definition of an operational semantics $\mathbb{C}, m \rightarrow^{*} \mathbb{C}', m'$, where $m, m' \in \mathbb{M}$, and a reification function that relates a CoLoSL world to a concrete machine state.

Definition 9 (Reification). *The* reification, $\lfloor . \rfloor_{\mathsf{W}} : \mathsf{World} \rightarrow \mathbb{M}$ *is defined as:*

$$\lfloor ((\sigma_l, h_l, \kappa_l), (\sigma_g, h_g, \kappa_g), \mathcal{I}) \rfloor_{\mathsf{W}} \triangleq (\sigma_l \uplus \sigma_g, h_l \uplus h_g)$$

Definition 10 (Valid triple). *A triple is* valid, *written* $\vDash \{P\}\ \mathbb{C}\ \{Q\}$, *if and only if, for all* $\iota \in \mathsf{LEnv}$, $w \in \mathsf{World}$ *and* $m, m' \in \mathbb{M}$,
$(w, \iota \vDash P$ *and* $\mathbb{C}, \lfloor w \rfloor_{\mathsf{W}} \rightarrow^{*} \mathrm{skip}, m')$ *implies* $\exists w'.\, w', \iota \vDash Q$ *and* $m' = \lfloor w' \rfloor_{\mathsf{W}}$

Theorem 1 (Soundness). *If* $\vdash \{P\}\ \mathbb{C}\ \{Q\}$ *then* $\vDash \{P\}\ \mathbb{C}\ \{Q\}$.

4 Examples

4.1 Concurrent Spanning Tree

Programs manipulating arbitrary graphs present a significant challenge for compositional reasoning, because deep, *unspecified* sharing between different components of a graph results in changes to one subgraph affecting other subgraphs pointing into it. This makes it hard to reason about updates to each subgraph in isolation. In a concurrent setting, this difficulty is compounded by the fact that threads working on different parts of the graph may affect each other in ways that are difficult to reason about locally to each subgraph. Using a concurrent spanning tree algorithm, we demonstrate that CoLoSL reasoning might be just the right approach, as subjective views naturally provide arbitrary overlapping views of the shared state where interferences can be naturally tailored to a given subjective

view. With CoLoSL, we have achieved local reasoning about the shared state for this challenging program.

Our example, presented in Fig. 8, operates on a *directed binary graph* (henceforth simply *graph*): that is, a directed graph where each node has at most two successors, called its left and right children. The program concurrently computes an in-place spanning tree of the graph (i.e. a tree that covers all nodes of the graphs from a given root), as follows: each time a new node is encountered, two new threads are created that each prune the edges of its left and right children recursively. A mark bit is associated with each node to keep track of which nodes have already been visited. Each thread returns whether it marked the node it was called on itself or whether somebody else did. In the latter case, the parent thread removes the link from its own root node to the corresponding child. Intuitively, it is allowed to do so because the child has already been reached via some other path in the graph since it was marked by another thread.

We will prove that, given a shared graph as input, the program always returns a tree, i.e. all sharing and cycles have been appropriately removed. Pleasingly, the CoLoSL specification captures just the subgraph manipulated by the thread, instead of the whole graph.

To reason about this program, following [14] we use two representations of graphs. The first is a mathematical representation $\gamma = (V, E)$ where V is a finite set of vertices and $E : V \to (V \uplus \{\text{null}\}) \times (V \uplus \{\text{null}\})$ is a function associating each vertex with at most two successors, where null denotes the absence of a child. We write $n \in \gamma$ for $n \in V$, $\gamma(n)$ for $E(n)$ which also assumes $n \in \gamma$, $n \rightsquigarrow_\gamma n'$ for $n' \in \gamma(n)$, and $\rightsquigarrow_\gamma^*$ for the reflexive and transitive closure of $\rightsquigarrow_\gamma$.

Mathematical graphs are connected to their in-memory representations by a predicate $\mathsf{graph}\,(x, \gamma)$ shown in Fig. 6, denoting a spatial (in-heap) graph rooted at address x corresponding to the mathematical graph γ. This predicate uses the overlapping conjunction to account for potential sharing between the left and right children and for potential cycles in the graph. Each vertex is represented as three consecutive cells in the heap tracking the mark bit and the addresses of the left and right subgraphs. We write $x \mapsto m, l, r$ for $x \mapsto m * x + 1 \mapsto l * x + 2 \mapsto r$, and $x.l$ and $x.r$ for $x + 1$ and $x + 2$, respectively. When vertex x is in the unmarked state $\mathsf{U}\,(x, l, r)$, the whole cell $x \mapsto 0, l, r$ resides in the shared state. In the marked state $\mathsf{M}\,(x)$, only $x \mapsto 1$ is owned. In both cases, the shared state also contains the left and right subgraphs represented by $\mathsf{G}\,(l, \gamma)$ and $\mathsf{G}\,(r, \gamma)$.

To understand the difference in ownership between $\mathsf{U}\,(x, l, r)$ and $\mathsf{M}\,(x)$, let us look at the interference associated with the graph, which is the union of interferences pertaining each vertex in the graph. For each vertex $n \in \gamma$, the only permitted action is that of marking n. (For simplicity, this action does not require any capability.) When changing the mark field of n from 0 to 1, the current thread also claims ownership of its left and right pointers. Indeed, we observe that other threads need not access the children of n once they see that it has already been marked. The atomic CAS (compare-and-swap) instruction prevents two threads from concurrently marking the same node and claiming ownership of the same resource.

$$\mathsf{graph}\,(x,\gamma) \triangleq \boxed{\mathsf{G}\,(x,\gamma)}_{I_\gamma} \qquad I_\gamma \triangleq \bigcup_{n\in\gamma} I(n) \qquad I(n) \triangleq \{[\emptyset] : \exists l, r.\, \mathsf{U}\,(n,l,r) \rightsquigarrow \mathsf{M}\,(n)\}$$

$$\mathsf{G}\,(x,\gamma) \triangleq (x = \mathrm{null} \wedge \mathsf{emp}) \vee \exists l, r.\, \gamma(x) = (l,r) \wedge (\mathsf{U}\,(x,l,r) \vee \mathsf{M}\,(x)) \curlyvee \mathsf{G}\,(l,\gamma) \curlyvee \mathsf{G}\,(r,\gamma)$$

$$\mathsf{U}\,(x,l,r) \triangleq x \mapsto 0,l,r \qquad \mathsf{M}\,(x) \triangleq x \mapsto 1$$

Fig. 6. Globally-shared graph predicate

$$\mathsf{g}\,(x,\gamma) \triangleq (x = \mathrm{null} \wedge \mathsf{emp}) \vee \exists l, r.\, \gamma(x) = (l,r) \wedge \boxed{\mathsf{U}\,(x,l,r) \vee \mathsf{M}\,(x)}_{I(x)} * \mathsf{g}\,(l,\gamma) * \mathsf{g}\,(r,\gamma)$$

$$\mathsf{t}\,(x,\gamma) \triangleq (x = \mathrm{null} \wedge \mathsf{emp}) \vee (\exists l, r.\, \gamma(x) = (l,r) \wedge \exists l' \in \{l, \mathrm{null}\}, r' \in \{r, \mathrm{null}\}.\ \boxed{\mathsf{M}\,(x)}_{I(x)} * x.l \mapsto l' * \mathsf{t}\,(l',\gamma) * x.r \mapsto r' * \mathsf{t}\,(r',\gamma))$$

Fig. 7. Locally-shared graph predicate

The $\mathsf{graph}\,(x,\gamma)$ predicate defined in Fig. 6 is a *global* subjective view of the graph that contains all vertices and the interference associated with γ. However, our spanning tree algorithm operates *locally* as it is called upon recursively for each node. That is, for each span(n) call (where $\mathrm{n} \in \gamma$), the footprint of the call is limited to the subgraph rooted at n. Moreover, in order to reason about the concurrent recursive calls span(x->l) || span(x->r), we need to *split* the state into two using $*$, pass each constituent state to the relevant thread and $*$-combine the resulting states, as required by the Par rule. We thus conduct our proof with respect to a *local* description of the graph, $\mathsf{g}\,(x,\gamma)$ as defined in Fig. 7. The definition of the $\mathsf{g}\,(x,\gamma)$ predicate is similar to that of $\boxed{\mathsf{G}\,(x,\gamma)}_{I_\gamma}$ except that the single view $\boxed{\mathsf{G}\,(x,\gamma)}_{I_\gamma}$ has been broken down into individual views for each vertex n reachable from x. Moreover, the interference assertion of each local view concerning a vertex $n \in \gamma$ has been shifted from I_γ to $I(n)$ so as to reflect only those actions that affect n.

The $\mathsf{t}\,(x,\gamma)$ predicate, also given in Fig. 7, represents a *tree* rooted at x, as is standard in separation logic [21], and consists, once fully unfolded, of one subjective view for each vertex n reachable from x in γ. The assertion of the subjective view for x reflects the fact that x has been marked, and its left and right pointers have been claimed by the marking thread and moved into its local state. The vertex l' addressed by the left pointer of x corresponds to either the initial value l prior to marking, or to null when l has more than one predecessor and has been marked by another thread.

Our goal is to prove that the following specification holds using the global graph predicate:

$$\{\mathsf{graph}\,(\mathrm{x},\gamma)\}\ \mathrm{b} = \mathrm{span(x)}\ \{(\mathrm{b} = 0 \wedge \mathsf{emp}) \vee (\mathrm{b} = 1 \wedge \mathsf{t}\,(\mathrm{x},\gamma))\} \tag{1}$$

This is achieved below by giving a proof of the analogous specification using the local graph predicate in Fig. 8, and demonstrating that the global graph specification implies the local one using the principles of CoLoSL from Fig. 1. The proof sketched in Fig. 8 is mostly straightforward. One subtlety is the consequence step

```
b = span(x) //{g(x,γ)}
{ //{(x = null ∧ emp) ∨ (∃l,r. γ(x) = (l,r) ∧ [U(x,l,r) ∨ M(x)]_I(x) * g(l,γ) * g(r,γ))}
  if (x == null) {
    //{x = null ∧ emp}
    //{t(x,γ)}
    return 1;
  }//{∃l,r. γ(x) = (l,r) ∧ [U(x,l,r) ∨ M(x)]_I(x) * g(l,γ) * g(r,γ)}
  res = ⟨ CAS(x, 0, 1) ⟩;
  //{∃l,r. γ(x) = (l,r) ∧ [M(x)]_I(x) * g(l,γ) * g(r,γ) *
  //  ((res = 0 ∧ emp) ∨ (res = 1 ∧ x.l ↦ l * x.r ↦ r))}
  //{(res = 0 ∧ emp) ∨ (res = 1 ∧ ∃l,r. γ(x) = (l,r) ∧
  //  [M(x)]_I(x) * g(l,γ) * g(r,γ) * x.l ↦ l * x.r ↦ r)}
  if (res) {
    //{res = 1 ∧ ∃l,r. γ(x) = (l,r) ∧ [M(x)]_I(x) * g(l,γ) * g(r,γ) * x.l ↦ l * x.r ↦ r}
      //{g(l,γ)}                                  ||  //{g(r,γ)}
      b1 = span(x->l);                            ||  b2 = span(x->r);
      //{(b1 = 0 ∧ emp) ∨ (b1 = 1 ∧ t(l,γ))}     ||  //{(b2 = 0 ∧ emp) ∨ (b2 = 1 ∧ t(r,γ))}
    //{res = 1 ∧ ∃l,r. γ(x) = (l,r) ∧ [M(x)]_I(x) * x.l ↦ l * x.r ↦ r *
    //  ((b1 = 0 ∧ emp) ∨ (b1 = 1 ∧ t(l,γ))) * ((b2 = 0 ∧ emp) ∨ (b2 = 1 ∧ t(r,γ)))}
    if (!b1) { x->l = null; }
    if (!b2) { x->r = null; }
    //{res = 1 ∧ ∃l,r. γ(x) = (l,r) ∧ [M(x)]_I(x) *
    //  ∃l' ∈ {l, null}. x.l ↦ l' * t(l',γ) * ∃r' ∈ {r, null}. x.r ↦ r' * t(r',γ)}
    //{res = 1 ∧ t(x,γ)}
  } //{(res = 0 ∧ emp) ∨ (res = 1 ∧ t(x,γ))}
  return res;
} //{(b = 0 ∧ emp) ∨ (b = 1 ∧ t(x,γ))}
```

Fig. 8. Code and proof sketch of the concurrent spanning tree program. We omit the obvious variables as resource assertions.

just before the second **if** statement. There, we first distribute the shared state $\boxed{\mathsf{M}(x)}_{I(x)} * \mathsf{g}(l,\gamma) * \mathsf{g}(r,\gamma)$ over the disjunction, then use the fact that it implies emp to discard it in the left disjunct. This proof demonstrates that our CoLoSL reasoning really is compositional, in the sense that we are doing local reasoning on the shared state (the subgraphs).

Let us show how to derive the specification (1) from the one obtained in Fig. 8. We introduce the *iterative star* operator $\circledast$; when iterating over the empty set, it denotes emp by convention (needed below, when x is null). We define

$$P(x,\gamma) \triangleq \circledast_{x \rightsquigarrow^*_\gamma n} \exists l,r.\, \gamma(n) = (l,r) \wedge (\mathsf{U}(n,l,r) \vee \mathsf{M}(n))$$

$$Q(x,\gamma) \triangleq \circledast_{x \rightsquigarrow^*_\gamma n} \exists l,r.\, \gamma(n) = (l,r) \wedge \boxed{\mathsf{U}(n,l,r) \vee \mathsf{M}(n)}_{I(n)}$$

From the definitions of $\mathsf{G}(x,\gamma)$ and $\mathsf{g}(x,\gamma)$, one can show that

$$\mathsf{graph}(x,\gamma) \iff \boxed{P(x,\gamma)}_{I_\gamma} \qquad \mathsf{g}(x,\gamma) \iff Q(x,\gamma)$$

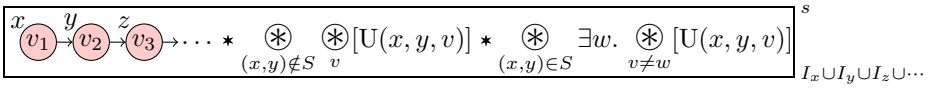
The specification (1) then follows from Consq, Fig. 8, and the derivation below:

$$\mathsf{graph}\,(x,\gamma) \overset{(\textsc{Copy})}{\Rightarrow} \circledast_{x \rightsquigarrow^*_\gamma n} \boxed{P(x,\gamma)}_{I_\gamma}$$
$$\overset{(\textsc{Forget})}{\Rightarrow} \circledast_{x \rightsquigarrow^*_\gamma n} \left(\exists l, r.\, \gamma(n) = (l,r) \wedge \boxed{\mathsf{U}\,(n,l,r) \vee \mathsf{M}\,(n)}_{I_\gamma}\right)$$
$$\overset{(\textsc{Shift})}{\Rightarrow} \circledast_{x \rightsquigarrow^*_\gamma n} \left(\exists l, r.\, \gamma(n) = (l,r) \wedge \boxed{\mathsf{U}\,(n,l,r) \vee \mathsf{M}\,(n)}_{I(n)}\right) \overset{\text{def}}{\Longleftrightarrow} \mathsf{g}\,(x,\gamma)$$

4.2 Set Module

Finally, we give a pictorial description of our reasoning about a concurrent set module implemented as a singly-linked list. We compare our CoLoSL reasoning with the original CAP reasoning of [7], demonstrating that our CoLoSL reasoning provides more concise proofs using our local reasoning about shared state.

Consider the following diagram which illustrates the CAP set predicate of [7]:



The set is represented as a sorted singly-linked list with no duplicate elements. The list starts at address x with value v_1, points to the next element at address y with value v_2, and so forth. Hereafter, we write $\mathsf{node}\,(x,v,y)$ to denote a node at address x, with value v and successor y.

All nodes of the list reside in a single shared region labelled s and the interference on the list is the combined interference associated with each constituent node. Each node at a given address x is associated with a set of update capabilities of the form $[\mathrm{U}(x,y,v)]$ for *all* possible addresses y and *all* possible values v. This is to capture all potential successor addresses y and all potential values v that may be stored at address x. In order to modify a node, a thread can acquire the lock associated with the node and subsequently claim the relevant update capability.

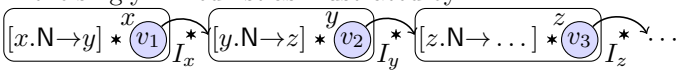
Since in CAP the capabilities associated with a region can only be generated upon its creation, the shared region is required to keep track of all possible update capabilities $[\mathrm{U}(x,y,v)]$ associated with all addresses x (including those not currently in the domain of the list), all addresses y and all values v. At any one point, given $\mathsf{node}\,(x,v,y)$, the only update capability that can be claimed by a thread (through locking) is the one that reflects its current status, namely $[\mathrm{U}(x,y,v)]$. As a result, an auxiliary mathematical set S is used to track those nodes of the list that are currently locked and thus infer which [U] capabilities have been claimed. The distribution of update capabilities is captured by the two assertions written as the *infinite multiplicative star operator* $\circledast$. The first part of the assertion states that given any node at address x with successor y, if it is not locked, i.e. $(x,y) \notin S$, then all of its update capabilities of the form $[\mathrm{U}(x,y,v)]$ lie in the shared region for all values v. Dually, if it is locked, i.e. $(x,y) \in S$, then the update capabilities for all values v but one $(w \neq v)$ are in the shared region.

This CAP set predicate is unnecessarily complicated. It is counter-intuitive to have to account for the capabilities associated with addresses not in the domain of the list. Moreover, each thread observes all nodes in the list and thus needs to account for their associated interference.

TaDA [3] took the first steps towards addressing the above shortcomings of the CAP approach. TaDA regions are parametric in the separation algebra of capabilities (called guards). As such, one can choose a more suitable algebra to axiomatise the desired behaviour of capabilities. While TaDA's approach is much cleaner than that of CAP, it nevertheless requires the foresight of specifying all desired interference associated with the region upon its creation. As such, interference specifications are *static* and cannot be extended with new behaviour even when the existing resources are left untouched. On the other hand, as well as being parametric in its capability separation algebra, the dynamic subjective views of CoLoSL provide local reasoning about the shared resource and its interference.

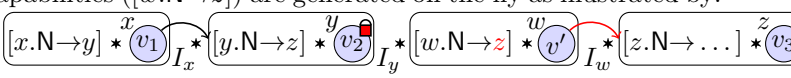
We proceed with the CoLoSL proof of the set implementation. Recall from §3 that CoLoSL is parametric in the separation algebra of capabilities. We thus instantiate it with a heap-like capability separation algebra that is *stateful* and demonstrate that this allows for a more concise proof.

We specify the set predicate as the $*$-composition of subjective views associated each node in the singly-linked list as illustrated by:



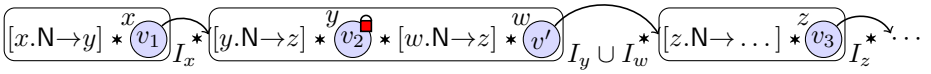
The interference on each subjective view is limited to the node in question. Associated with each node at address x is a "next" capability $[x.\mathsf{N}{\to}y]$ that tracks its successor y. This is analogous to the $[\mathrm{U}(x, y, v)]$ capability of CAP and we shortly demonstrate how it is utilised in our reasoning.

Since CoLoSL allows for dynamic extension of the shared state, we do not need to account for capabilities associated with all addresses. Instead, fresh capabilities are generated dynamically as needed. We demonstrate this by giving a reasoning outline of the add(v') method that adds value v' to the set by inserting it in the sorted list. Suppose $v_2 < v' < v_3$, and thus a new node w with value v' is to be inserted after node y. The operating thread proceeds by traversing the list by hand-over-hand locking until it reaches node y. It then locks y and claims its next pointer and moves it to its local state, as allowed by I_y. Subsequently, the shared state is extended by the resources associated with the new node and its associated capabilities ($[w.\mathsf{N}{\to}z]$) are generated on the fly as illustrated by:

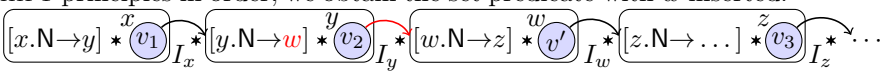


Since the locking thread holds the next pointer of y in its local state, it modifies it to point to the new node w. It then unlocks y and returns its next pointer to the shared state. When inserting a new node between y and z, the associated interference assertion I_y allows y to be unlocked only if it has been directed to a new node whose successor is z. As such, the unlocking thread must demonstrate

that the new node w does indeed point to z. In order to establish this, we use the MERGE principle to combine the subjective views of y and w as follows:



Finally, y is unlocked; its next pointer is returned to the shared state and its next capability is modified to reflect its new successor. Using the COPY, FORGET and SHIFT principles in order, we obtain the set predicate with w inserted.



We can reason about the remove operation in a similar fashion. The dynamic extension afforded by the EXTEND principle allows us to generate new capabilities only when needed and thus gives way to a concise proof. Moreover, rather than having a distinct capability to modify the element at address x, for each possible successor address y (as with $[\mathrm{U}(x, y, v)]$ in CAP), we appeal to a single capability of the form $[x.\mathsf{N}{\rightarrow}y]$ that is modified to $[x.\mathsf{N}{\rightarrow}y']$ whenever x's successor changes from y to y'. Lastly, using the reasoning principles of MERGE, FORGET, SHIFT and COPY, we can grow and shrink our subjective views as needed. This means that, at any one point, we only view the relevant parts of the shared state. The technical details can be found in [20].

Concluding Remarks. We have introduced CoLoSL, a new program logic for reasoning locally about the shared state. We focus on subjective views, which expand and contract to provide a flexible treatment of both the shared resource and its interference. However, CoLoSL is still young, and lacks many features of its various cousins. There are many interesting ideas present in the literature: e.g. abstract states governed by state transition systems [25]; higher-order reasoning [23]; and abstract atomicity [3]. All these ideas require further investigation. Here, our aim was to simply introduce subjective views as a fundamental new way of underpinning such reasoning.

Acknowledgements. We are grateful to Aquinas Hobor for providing us with the example of §2, and to Matthew Parkinson for suggesting to record catalysts in the model. We would also like to thank Pedro da Rocha Pinto for his continuous feedback on earlier versions of this paper. This research was funded by EPSRC grants K008528/1 and H008373/2.

References

1. Bornat, R., Calcagno, C., Yang, H.: Variables as resource in separation logic. ENTCS 155 (2006)
2. Calcagno, C., O'Hearn, P.W., Yang, H.: Local action and abstract separation logic. In: LICS (2007)
3. da Rocha Pinto, P., Dinsdale-Young, T., Gardner, P.: TaDA: A logic for time and data abstraction. In: Jones, R. (ed.) ECOOP 2014. LNCS, vol. 8586, pp. 207–231. Springer, Heidelberg (2014)
4. Dijkstra, E.: A belated proof of self-stabilization. Distributed Computing 1(1) (1986)

5. Dijkstra, E.W.: Self-stabilizing systems in spite of distributed control. Commun. ACM 17(11) (1974)
6. Dinsdale-Young, T., Birkedal, L., Gardner, P., Parkinson, M.J., Yang, H.: Views: compositional reasoning for oncurrent programs. In: POPL (2013)
7. Dinsdale-Young, T., Dodds, M., Gardner, P., Parkinson, M.J., Vafeiadis, V.: Concurrent abstract predicates. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 504–528. Springer, Heidelberg (2010)
8. Dockins, R., Hobor, A., Appel, A.W.: A fresh look at separation algebras and share accounting. In: Hu, Z. (ed.) APLAS 2009. LNCS, vol. 5904, pp. 161–177. Springer, Heidelberg (2009)
9. Dodds, M., Feng, X., Parkinson, M., Vafeiadis, V.: Deny-guarantee reasoning. In: Castagna, G. (ed.) ESOP 2009. LNCS, vol. 5502, pp. 363–377. Springer, Heidelberg (2009)
10. Feng, X.: Local rely-guarantee reasoning. In: POPL. ACM (2009)
11. Gardner, P., Maffeis, S., Smith, G.D.: Towards a program logic for JavaScript. In: POPL. ACM (2012)
12. Gardner, P., Maffeis, S., Smith, G.D.: Towards a program logic for JavaScript. In: POPL. ACM (2012)
13. Gardner, P., Raad, A., Wheelhouse, M., Wright, A.: Abstract local reasoning for concurrent libraries: mind the gap. In: MFPS (2014)
14. Hobor, A., Villard, J.: The ramifications of sharing in data structures. In: POPL (2013)
15. Ishtiaq, S.S., O'Hearn, P.W.: BI as an assertion language for mutable data structures. In: POPL (2001)
16. Jones, C.B.: Specification and design of (parallel) programs. In: IFIP Cong (1983)
17. Nanevski, A., Ley-Wild, R., Sergey, I., Delbianco, G.A.: Communicating state transition systems for fine-grained concurrent resources. In: Shao, Z. (ed.) ESOP 2014 (ETAPS). LNCS, vol. 8410, pp. 290–310. Springer, Heidelberg (2014)
18. O'Hearn, P.W.: Resources, concurrency, and local reasoning. TCS 375(1-3) (2007)
19. O'Hearn, P.W., Reynolds, J.C., Yang, H.: Local reasoning about programs that alter data structures. In: Fribourg, L. (ed.) CSL 2001 and EACSL 2001. LNCS, vol. 2142, p. 1. Springer, Heidelberg (2001)
20. Raad, A., Villard, J., Gardner, P.: CoLoSL: Concurrent Local Subjective Logic (2014), http://www.doc.ic.ac.uk/~azalea/ESOP2015/CoLoSL-TR.pdf
21. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: LICS. IEEE Computer Society (2002)
22. Reynolds, J.C.: A short course on separation logic (2003), http://www.cs.cmu.edu/afs/cs.cmu.edu/project/fox-19/member/jcr/wwwaac2003/notes7.ps
23. Svendsen, K., Birkedal, L.: Impredicative concurrent abstract predicates. In: Shao, Z. (ed.) ESOP 2014 (ETAPS). LNCS, vol. 8410, pp. 149–168. Springer, Heidelberg (2014)
24. Svendsen, K., Birkedal, L., Parkinson, M.: Modular reasoning about separation of concurrent data structures. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 169–188. Springer, Heidelberg (2013)
25. Turon, A., Dreyer, D., Birkedal, L.: Unifying refinement and hoare-style reasoning in a logic for higher-order concurrency. SIGPLAN Not 9 (2013)
26. Vafeiadis, V., Parkinson, M.: A marriage of rely/Guarantee and separation logic. In: Caires, L., Vasconcelos, V.T. (eds.) CONCUR 2007. LNCS, vol. 4703, pp. 256–271. Springer, Heidelberg (2007)