Verifying Concurrent Graph Algorithms

Azalea Raad[†], Aquinas Hobor[‡], Jules Villard[†], Philippa Gardner[†]

[†]Imperial College London [‡]Yale-NUS College and School of Computing, National University of Singapore

Abstract. We show how to verify four challenging concurrent fine-grained graph-manipulating algorithms, including graph copy, a speculatively parallel Dijkstra, graph marking and spanning tree. We develop a reasoning method for such algorithms that dynamically tracks the contributions and responsibilities of each thread operating on a graph, even in cases of arbitrary recursive thread creation. We demonstrate how to use a logic without abstraction (CoLoSL) to carry out abstract reasoning in the style of iCAP, by building the abstraction into the proof structure rather than incorporating it into the semantic model of the logic.

1 Introduction

The verification of fine-grained concurrent algorithms is nontrivial. There has been much recent progress verifying such algorithms modularly using variants of concurrent separation logic [9, 12, 16, 15, 6, 4]. One area of particular difficulty has been verifying such algorithms that manipulate graphs. This is only to be expected: even in a semi-formal "algorithmic" sense, the correctness arguments of concurrent graph algorithms can be dauntingly subtle [2].

To verify such algorithms, we must not only understand these algorithmic arguments, but must also determine a precise way to express them in a suitable formal system. Even sequential graph algorithms are challenging to verify due to the overlapping nature of the graph structures, preventing e.g. easy use of the frame rule of separation logic [8]. Concurrent graph algorithms pose a number of additional challenges, such as reasoning how the actions of each thread advance the overall goal despite the possible interference from other threads. Unsurprisingly, verifications of such algorithms are rare in the literature.

We verify the functional correctness of four nontrivial concurrent fine-grained graph algorithms. We study a structure-preserving copy, a speculatively-parallel version of Dijkstra's shortest-path algorithm, a graph marking, and a spanning tree algorithm. We have found common "proof patterns" for tackling these algorithms, principally reasoning about the functional correctness of the algorithm on abstract *mathematical* graphs γ, defined as sets of vertices and edges. We use such abstractions to state and prove key invariants. Another pattern is that we track the progress of each thread using a notion of *tokens* to record each thread's portion of the computation. Informally, if the token of thread t is on vertex v, then t is responsible for some work on/around v. Our tokens are sufficiently general to handle sophisticated parallelism. (e.g. dynamic thread creation/destruction).

We then reason about the memory safety of the algorithm by connecting our reasoning on mathematical graphs to *spatial* graphs (sets of memory cells in the heap) by defining spatial predicates that implement mathematical structures in the heap e.g. $\mathsf{graph}(\gamma) \overset{\text{def}}{=} \ldots$. We define our spatial predicates in such a way that simplifies many of the proof obligations (e.g. when parallel computations join).

This pattern of doing the algorithmic reasoning on abstract states is similar to the style of reasoning used in logics such as CaReSL [16] and iCAP [15]. CaReSL introduced the idea of reasoning on abstract states. Later, iCAP extended the program logic of CAP [4] to reason about higher-order code and adopted CaReSL's abstract states. Just as with these logics, we carry out our reasoning on abstract states, which enables simpler proofs and lessens the burden of side conditions such as establishing stability. With these logics, this abstract style of reasoning has been "baked in" to the semantic models. Here, we demonstrate that this baking is unnecessary by using a logic (CoLoSL [12]) without such built-in support. We do not use any of the unique features of CoLoSL. As such, we believe that our proofs and style of abstract reasoning port to other program logics without difficulty.

Related work There has been much work on reasoning about graph algorithms using separation logic. For *sequential* graph algorithms, Bornat et al. presented preliminary work on dags in [1], Yang studied the Schorr-Waite graph algorithm [17], Reynolds conjectured how to reason about dags [13], and Hobor and Villard showed how to reason about dags and graphs [8]. We make critical use of some of Hobor and Villard's graph-related verification infrastructure.

Many *concurrent* program logics have been proposed in recent years; both iCAP and CaReSL encourage the kind of abstract reasoning we employ in our verifications. However, published examples in these logics focus heavily on verifying concurrent data structures, whereas we focus on verifying concurrent graph algorithms. Moreover, the semantic models for both of these logics incorporate significant machinery to enable this kind of abstract reasoning, whereas we are able to use it without built-in support.

There has hardly been any work on *concurrent* graph algorithms. Raad et al. [12] and Sergey et al. [14] have verified a concurrent spanning tree algorithm, one of our examples. In [12], Raad et al. introduced CoLoSL and gave a shaped-based proof of spanning tree to demonstrate CoLoSL reasoning. A full functional correctness proof in CoLoSL was available at the time, although not using the proof pattern presented here. Later in [14], Sergey et al. gave a full functional correctness proof in Coq, but only that single example. We believe we are the first to verify `copy_dag`, which is known to be difficult, and `parrellel_dijkstra`, which we believe is the first verification of an algorithm that uses speculative parallel decomposition [7].

Outline The rest of this paper is organised as follows. In §2 we give an overview of the CoLoSL program logic and outline our proof pattern. We then use our proof pattern to verify the concurrent `copy_dag` (§3) and `parallel_dijkstra` (§4) algorithms, and finish with concluding remarks. We refer the reader to [10, 11] for the verification of two further concurrent graph algorithms for graph marking and computing the spanning tree of a graph.

2 Background

2.1 CoLoSL: Concurrent Local Subjective Logic

In the program logic of CoLoSL [12], the program state is modelled as a pair comprising a *thread-local* state, and a *global shared* state accessible by all threads. For instance, a shared counter at location x can be specified as:

$$\mathsf{C} \overset{\text{def}}{=} \iota * \boxed{\exists v {\leqslant} max.\, x \mapsto v * x{+}1 \mapsto max}_{I} \qquad I \overset{\text{def}}{=} \left\{\iota \colon x \mapsto v \land v {<} max \;\rightsquigarrow\; x \mapsto v{+}1\right.$$

The assertion C states that the counter at location x is a *shared* resource (denoted by the $\boxed{\text{box}}$) with some value $v \leqslant max$, that the maximum value permitted for the counter (max) is also a shared resource stored at location $x{+}1$, and that the current thread holds some *capability* ι in its *local* state. The *interference* relation, I, describes how the shared state may be updated and is specified through actions indexed by capabilities. A thread can perform an action if it holds the capability for that action in its local state. Here, I declares one action for incrementing the value of x, indexed by the increment capability ι. As such, this thread (or any other that also holds some ι capability in its local state) may increment x by one, provided that the incremented value does not exceed max.

Shared state assertions can be freely duplicated using the Copy principle in Fig. 1. This allows us to duplicate and pass on the knowledge about the shared state to new threads, using the standard parallel composition rule Par. To allow local reasoning, a thread may weaken its view of the shared state to obtain a partial *subjective* view of it using the Forget principle. For instance given the counter specification C above, if this thread is not interested in location $x{+}1$ where max is stored, it may *forget* it and obtain $\boxed{\exists v \leqslant max.\, x \mapsto v}_{I}$. That is, each (subjective) shared state assertion describes (potentially) only parts of the shared global resources. As such, subjective views may arbitrarily overlap with each other. For instance, while this thread chooses to forget the $x{+}1$ location in C, a second thread may choose to observe both x and $x{+}1$, and a third thread may choose to observe $x{+}1$ only. CoLoSL also allows for weakening (framing) of the interference relation using the Shift principle: $\boxed{P}_{I \cup I'} \land$ [side-condition-omitted] $\overset{\text{Shift}}{\Longrightarrow} \boxed{P}_{I}$. Hence, subjective views may also arbitrarily overlap in their interference relations. Due to space constraints we have omitted this rule from Fig. 1 as we do not use it in this paper. Different subjective views of the shared state can be combined using the Merge principle. Since subjective views may overlap both in their resources and interference relations, we use the *overlapping conjunction* [8], $\circledast$, to combine the resources, and set union $\cup$ to combine their interference relations. Intuitively, $P \circledast Q$ describes a state comprising two (potentially) overlapping parts satisfying P and Q, respectively.

CoLoSL is parametric in the model of its resources and may be instantiated with any PCM (partial commutative monoid).[1] In the example above (counter), our resource PCM is that of ordinary concrete heaps, $\mathbb{H} \overset{\text{def}}{=} (\mathcal{H}, \uplus, \varnothing)$, with the

[1] CoLoSL stipulates that PCMs satisfy the cross-split property [8], which ours do.

$$\boxed{P}_I \overset{\text{Copy}}{\Longrightarrow} \boxed{P}_I \ast \boxed{P}_I \qquad \boxed{P \ast Q}_I \overset{\text{Forget}}{\Longrightarrow} \boxed{P}_I \qquad \boxed{P}_{I_1} \ast \boxed{Q}_{I_2} \overset{\text{Merge}}{\Longrightarrow} \boxed{P \circledast Q}_{I_1 \cup I_2}$$

$$\frac{\{P_1\}\ \texttt{C1}\ \{Q_1\} \quad \{P_2\}\ \texttt{C2}\ \{Q_2\}}{\{P_1 \ast P_2\}\ \texttt{C1}\,||\,\texttt{C2}\ \{Q_1 \ast Q_2\}}\ \textsc{Par} \qquad \frac{P \Rrightarrow P' \quad \{P'\}\ \texttt{C}\ \{Q'\} \quad Q' \Rrightarrow Q}{\{P\}\ \texttt{C}\ \{Q\}}\ \textsc{Con}$$

Figure 1: An excerpt of the reasoning principles and proof rules in CoLoSL

composition operator as the disjoint function union, and the function with empty domain ($\varnothing$) as the single unit element. In the remainder of this paper, we take our PCM elements as pairs (h_c, h_g) in the PCM $\mathbb{H}^2 \overset{\text{def}}{=} \big(\mathcal{H}^2, (\uplus, \uplus), (\varnothing, \varnothing)\big)$ where h_c is the concrete heap, and h_g is the ghost heap. CoLoSL is also parametric in its capability model and may be instantiated with any PCM. In the following sections, we choose the capability PCM on a per-example basis.[1]

CoLoSL borrows the consequence rule (Con) of the Views framework [3], with $\Rrightarrow$ denoting the *semantic consequence* relation (semantic implication). That is, we write $P \Rrightarrow Q$ when the set of low-level machine states described by P are contained in that of Q. This way ghost heaps may be manipulated by an application of Con rather than explicit ghost instructions.

2.2 Proof Pattern: Combining Mathematical and Spatial Reasoning

Our graph verifications follow a common pattern which we outline as follows. First, we select an appropriate abstract model for *mathematical graphs*, which is typically sets of vertices and edges together with labels. Second, we choose a *token* model. We use tokens to identify each thread uniquely and to track the contribution of each thread to the global computation. For instance, for an algorithm with only two threads this might be as simple as the set $\{\mathsf{red}, \mathsf{blue}\}$, identifying each thread as a distinct colour.

Third, we define *mathematical actions* to capture the operations performed by threads. These actions model both *concrete* updates to the graph (e.g. removing an edge), as well as *ghost* updates used solely for reasoning (e.g. adding or removing tokens to track the computation progress). Fourth, we define *mathematical assertions* to describe program invariants and pre-/postconditions. These assertions are on mathematical graphs and involve abstract concepts (e.g. reachability along a path). As a key proof obligation, we must prove that our mathematical assertions are *stable* with respect to our mathematical actions, i.e. they remain true under the actions of other threads in the environment.

Fifth, we define *spatial predicates* (e.g. $\mathsf{graph}(\gamma)$) that describe how mathematical graphs are implemented in the heap. For instance, a graph may be implemented as a set of heap-linked nodes or as an adjacency matrix. We then combine these spatial predicates with our mathematical actions to define *spatial actions*. Intuitively, if a mathematical action transforms γ to γ', then the corresponding spatial action transforms $\mathsf{graph}(\gamma)$ to $\mathsf{graph}(\gamma')$.

3 Copying Heap-represented Dags Concurrently

The `copy_dag(x)` program in Fig. 4 makes a deep structure-preserving copy of the dag (directed acyclic graph) rooted at `x` concurrently. To do this, each node `x` in the source dag records in its copy field (`x->c`) the location of its copy when it exists, or 0 otherwise. Our language is C with a few cosmetic differences. Line 1 gives the data type of heap-represented dags. The statements between angle brackets `<.>` (e.g. lines 5-7) denote atomic instructions that cannot be interrupted by other threads. We write `C1 || C2` (e.g. line 9) for the parallel computation of `C1` and `C2`. This corresponds to the standard fork-join parallelism.

A thread running `copy_dag(x)` first checks atomically (lines 5-7) if `x` has already been copied. If so, the address of the copy is returned. Otherwise, the thread allocates a new node `y` to serve as the copy of `x` and updates `x->c` accordingly; it then proceeds to copy the left and right subdags in parallel by spawning two new threads (line 9). At the beginning of the initial call, none of the nodes have been copied and all copy fields are 0; at the end of this call, all nodes are copied to a new dag whose root is returned by the algorithm. In the intermediate recursive calls, only parts of the dag rooted at the argument are copied. Note that the atomic block of lines 5-7 corresponds to a `CAS` (compare and set) operation. We have unwrapped the definition for better readability.

Although the code is short, its correctness argument is rather subtle as we need to reason simultaneously about both deep unspecified sharing inside the dag as well as the parallel behaviour. This is not surprising since the unspecified sharing makes verifying even the sequential version of similar algorithms non-trivial [8]. However, the non-deterministic behaviour of parallel computation makes even *specifying* the behaviour of `copy_dag` challenging. Observe that each node x of the source dag may be in one of the following three stages:

1. x is not visited by any thread (not copied yet), and thus its copy field is 0.
2. x has already been visited by a thread π, a copy node x' has been allocated, and the copy field of x has been accordingly updated to x'. However, the edges of x' have not been directed correctly. That is, the thread copying x has not yet finished executing line 10.
3. x has been copied and the edges of its copy have been updated accordingly.

Note that in stage 2 when x has already been visited by a thread π, if another thread π' visits x, it simply returns even though x and its children may not have been fully copied yet. How do we then specify the postcondition of thread π' since we cannot promise that the subdag at x is fully copied when it returns? Intuitively, thread π' can safely return because another thread (π) has copied x and has made a *promise* to visit its children and ensure that they are also copied (by which time the said children may have been copied by other threads, incurring further promises). More concretely, to reason about `copy_dag` we associate each node with a *promise set* identifying those threads that must visit it.

Consider the dags in Fig. 2 where a node x is depicted as i) a white circle when in stage 1, e.g. $\bigcirc\!\!\!\!\!\!\;{\scriptstyle x,0}$ in 2a; ii) a grey ellipse when in stage 2, e.g. $\frac{x,x'}{\pi}$ in

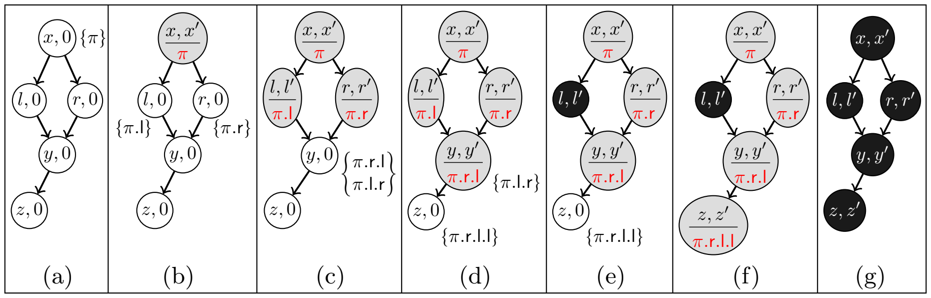
2b where thread π has copied x to x'; and iii) a black circle when in stage 3, e.g. x, x' in 2g. Initially no node is copied and as such all copy fields are 0. Let us assume that the top thread (the thread running the very first call to `copy_dag`) is identified as π. That is, thread π has made a promise to visit the top node x and as such the promise set of x comprises π. This is depicted in the initial snapshot of the graph in Fig. 2a by the $\{\pi\}$ promise set next to x. Thread π proceeds with copying x to x', and transforming the dag to that of Fig. 2b. In doing so, thread π fulfils its promise to x and π is thus removed from the promise set of x. Recall that if another thread now visits x it simply returns, relinquishing the responsibility of copying the descendants of x. This is because the responsibility to copy the left and right subdags of x lies with the left and right sub-threads of π (spawned at line 9), respectively. As such, in transforming the dag from Fig. 2a to 2b, thread π extends the promise sets of l and r, where $\pi.\mathsf{l}$ (resp. $\pi.\mathsf{r}$) denotes the left (resp. right) sub-thread spawned by π at line 9. Subsequently, the $\pi.\mathsf{l}$ and $\pi.\mathsf{r}$ sub-threads copy l and r as illustrated in Fig. 2c, each incurring a promise to visit y via their sub-threads. That is, since both l and r have an edge to y, they race to copy the subdag at y. In the trace detailed in Fig. 2, the $\pi.\mathsf{r}.\mathsf{l}$ sub-thread wins the race and transforms the dag to that of Fig. 2d by removing $\pi.\mathsf{r}.\mathsf{l}$ from the promise set of y, and incurring a promise at z. Since the $\pi.\mathsf{l}.\mathsf{r}$ sub-thread lost the race for copying y, it simply returns (line 3). That is, $\pi.\mathsf{l}.\mathsf{r}$ needs not proceed to copy y as it has already been copied. As such, the promise of $\pi.\mathsf{l}.\mathsf{r}$ to y is trivially fulfilled and the copying of l is finalised. This is captured in the transition from Fig. 2d to 2e where $\pi.\mathsf{l}.\mathsf{r}$ is removed from the promise set of y, and l is taken to stage 3. Thread $\pi.\mathsf{r}.\mathsf{l}.\mathsf{l}$ then proceeds to copy z, transforming the dag to that of Fig. 2f. Since z has no descendants, the copying of the subdag at z is now at an end; thread $\pi.\mathsf{r}.\mathsf{l}.\mathsf{l}$ thus returns, taking z to stage 3. In doing so, the copying of the entire dag is completed; sub-threads join and the effect of copying is propagated to the parent threads, taking the dag to that depicted in Fig. 2g.

Note that in order to track the contribution of each thread and record the overall copying progress, we must identify each thread uniquely. To this end, we appeal to a *token* (identification) mechanism that can 1) distinguish one token (thread) from another; 2) identify two distinct sub-tokens given any token, to reflect the new threads spawned at recursive call points; and 3) model a parent-child relationship to discern the spawner thread from its sub-threads. We model our tokens as a variation of the tree share algebra in [5] as described below.

Trees as tokens A tree token (henceforth a token), $\pi \in \mathit{\Pi}$, is defined by the grammar below as a binary tree with boolean leaves ($\circ$, $\bullet$), exactly one $\bullet$ leaf, and unlabelled internal nodes.

$$\mathit{\Pi} \ni \pi ::= \bullet \mid \widehat{\circ\ \pi} \mid \widehat{\pi\ \circ}$$

We refer to the thread associated with π as thread π. To model the parent-child relation between thread π and its two sub-threads (left and right), we define a mechanism for creating two distinct sibling tokens $\pi.\mathsf{l}$ and $\pi.\mathsf{r}$ defined below. Intuitively, $\pi.\mathsf{l}$ and $\pi.\mathsf{r}$ denote replacing the $\bullet$ leaf of π with $\widehat{\circ\ \bullet}$ and $\widehat{\bullet\ \circ}$, respectively. We model the ancestor-descendant relation between threads by the



$x,0$ $\{\pi\}$
$l,0$
$r,0$
$y,0$
$z,0$
(a)
x,x' π
$\{\pi.\mathsf{l}\}$
$\{\pi.\mathsf{r}\}$
(b)
l,l' $\pi.\mathsf{l}$
r,r' $\pi.\mathsf{r}$
$\{\pi.\mathsf{r}.\mathsf{l}, \pi.\mathsf{l}.\mathsf{r}\}$
(c)
y,y' $\pi.\mathsf{r}.\mathsf{l}$
$\{\pi.\mathsf{l}.\mathsf{r}\}$
$\{\pi.\mathsf{r}.\mathsf{l}.\mathsf{l}\}$
(d)
(e)
z,z' $\pi.\mathsf{r}.\mathsf{l}.\mathsf{l}$
(f)
(g)

Figure 2: An example trace of `copy_dag`

$\sqsubset$ ordering defined below where $+$ denotes the transitive closure of the relation.

$$\bullet.\mathsf{l} = \widehat{\circ\ \bullet} \qquad (\widehat{\circ\ \pi}).\mathsf{l} = \widehat{\circ\ \pi.\mathsf{l}} \qquad (\widehat{\pi\ \circ}).\mathsf{l} = \widehat{\pi.\mathsf{l}\ \circ}$$
$$\bullet.\mathsf{r} = \widehat{\bullet\ \circ} \qquad (\widehat{\circ\ \pi}).\mathsf{r} = \widehat{\circ\ \pi.\mathsf{r}} \qquad (\widehat{\pi\ \circ}).\mathsf{r} = \widehat{\pi.\mathsf{r}\ \circ} \qquad \sqsubset \overset{\text{def}}{=} \{(\pi.\mathsf{l}, \pi), (\pi.\mathsf{r}, \pi) \mid \pi \in \Pi\}^{+}$$

We write $\pi \sqsubseteq \pi'$ for $\pi{=}\pi' \vee \pi \sqsubset \pi'$, and write $\pi \not\sqsubset \pi'$ (resp. $\pi \not\sqsubseteq \pi'$) for $\neg(\pi \sqsubset \pi')$ (resp. $\neg(\pi \sqsubseteq \pi')$). Observe that $\bullet$ is the maximal token, i.e. $\forall \pi \in \Pi.\, \pi \sqsubseteq \bullet$. As such, the top-level thread is associated with the $\bullet$ token, since all other threads are its sub-threads and are subsequently spawned by it or its descendants (i.e. $\pi{=}\bullet$ in Figs. 2a-2g). In what follows we write $\overline{\pi}$ to denote the token set comprising the descendants of π, i.e. $\overline{\pi} \overset{\text{def}}{=} \{\pi' \mid \pi' \sqsubseteq \pi\}$.

As discussed in §2.2, we carry out most of our reasoning abstractly by appealing to *mathematical objects*. To this end, we define *mathematical dags* as an abstraction of the dag structure in `copy_dag`.

Mathematical dags A mathematical dag, $\delta \in \Delta$, is a triple of the form (V, E, L) where V is the vertex set; $E : V \to V_0 \times V_0$, is the edge function with $V_0{=}V \uplus \{0\}$, where 0 denotes the absence of an edge (e.g. a null pointer); and $L{=}V{\to}D$, is the vertex labelling function with the label set D defined shortly. We write δ^{V}, δ^{E} and δ^{L}, to project the various components of δ. Moreover, we write $\delta^{\mathsf{l}}(x)$ and $\delta^{\mathsf{r}}(x)$ for the first and second projections of $E(x)$; and write $\delta(x)$ for $(L(x), \delta^{\mathsf{l}}(x), \delta^{\mathsf{r}}(x))$ when $x \in V$. Given a function f (e.g. E, L), we write $f[x \mapsto v]$ for updating $f(x)$ to v, and write $f \uplus [x \mapsto v]$ for extending f with x and value v. Two dags are *congruent* if they have the same vertices and edges, i.e. $\delta_1 \cong \delta_2 \overset{\text{def}}{=} \delta_1^{\mathrm{V}}{=}\delta_2^{\mathrm{V}} \wedge \delta_1^{\mathrm{E}}{=}\delta_2^{\mathrm{E}}$. We define our mathematical objects as pairs of dags $(\delta, \delta') \in (\mathcal{W}_\delta \times \mathcal{W}_\delta)$, where δ and δ' denote the source dag and its copy, respectively.

To capture the stages a node goes through, we define the node labels as $D{=}\big(V_0 \times (\Pi \uplus \{0\}) \times \mathcal{P}(\Pi)\big)$. The first component records the *copy* information (the address of the copy when in stage 2 or 3; 0 when in stage 1). This corresponds to the second components in the nodes of the dags in Fig. 2, e.g. 0 in $\textcircled{x,0}$. The second component tracks the node *stage* as described on page 5: 0 in *stage* 1 (white nodes in Fig. 2), some π in stage 2 (grey nodes in Fig. 2), and 0 in stage 3 (black nodes in Fig. 2). That is, when the node is being *processed* by thread π, this component reflects the thread's token. Note that this is a *ghost* component in that it is used purely for reasoning and does not appear in the physical memory. The third (ghost) component denotes the *promise* set of the node and tracks the tokens of those threads that are yet to visit it. This corresponds to the sets adjacent to nodes in the dags of Fig. 2, e.g. $\{\pi.l\}$ in Fig. 2b. We write $\delta^{\mathsf{c}}(x)$, $\delta^{\mathsf{s}}(x)$ and $\delta^{\mathsf{p}}(x)$ for the first, second, and third projections of x's label, respectively. We define the *path* relation, $x \overset{\delta}{\rightsquigarrow} y$, and the *unprocessed path* relation, $x \overset{\delta}{\rightsquigarrow}_0 y$, as follows and write $\overset{\delta}{\rightsquigarrow}^{\star}$ and $\overset{\delta}{\rightsquigarrow}^{\star}_0$ for their reflexive transitive closure, respectively.

$$x \overset{\delta}{\rightsquigarrow} y \overset{\text{def}}{=} \delta^{\mathsf{l}}(x){=}y \vee \delta^{\mathsf{r}}(x){=}y \qquad\qquad x \overset{\delta}{\rightsquigarrow}_0 y \overset{\text{def}}{=} x \overset{\delta}{\rightsquigarrow} y \wedge \delta^{\mathsf{c}}(x) = 0 \wedge \delta^{\mathsf{c}}(y) = 0$$

The lifetime of a node x with label (c, s, P) can be described as follows. Initially, x is in stage 1 ($c{=}0$, $s{=}0$). When thread π visits x, it creates a copy node x' and takes x to stage 2 ($c{=}x'$, $s{=}\pi$). In doing so, it removes its token π from the promise set P, and adds $\pi.\mathsf{l}$ and $\pi.\mathsf{r}$ to the promise sets of its left and right children, respectively. Once π finishes executing line 10, it takes x to stage 3 ($c{=}x'$, $s{=}0$). If another thread π' then visits x when it is in stage 2 or 3, it removes its token π' from the promise set P, leaving the node stage unchanged.

As discussed in §2.2, to model the interactions of each thread π with the shared data structure, we define mathematical *actions* as relations on mathematical objects. We thus define several families of actions, each indexed by a token π.

Actions The mathematical actions of `copy_dag` are given in Fig. 3. The A^1_π describes taking a node x from stage 1 to 2 by thread π. In doing so, it removes its token π from the promise set of x, and adds $\pi.\mathsf{l}$ and $\pi.\mathsf{r}$ to the promise sets of its left and right children respectively, indicating that they will be visited by its sub-threads, $\pi.\mathsf{l}$ and $\pi.\mathsf{r}$. It then updates the copy field of x to y, and extends the copy graph with y. This action captures the atomic block of lines 5-7 when successful. The next two sets capture the execution of atomic commands in line 10 by thread π where A^2_π and A^3_π respectively describe updating the left and right edges of the copy node. Once thread π has finished executing line 10 (and has updated the edges of y), it takes x to stage 3 by updating the relevant ghost values. This is described by A^4_π. The A^5_π set describes the case where node x has already been visited by another thread (it is in stage 2 or 3 and thus its copy field is non-zero). Thread π then proceeds by removing its token from x's promise set. We write A_π to denote the actions of thread π: $A_\pi \overset{\text{def}}{=} A^1_\pi \cup A^2_\pi \cup A^3_\pi \cup A^4_\pi \cup A^5_\pi$. We can now specify the behaviour of `copy_dag` mathematically.

Mathematical specification Throughout the execution of `copy_dag`, the source dag and its copy (δ, δ'), satisfy the invariant Inv below.

$$\mathsf{Inv}(\delta,\delta') \overset{\text{def}}{=} \mathsf{acyc}(\delta) \wedge \mathsf{acyc}(\delta') \wedge (\forall x' \in \delta'.\, \exists! x \in \delta.\, \delta^{\mathsf{c}}(x){=}x') \wedge (\forall x \in \delta.\, \exists x'.\, \mathsf{ic}(x, x', \delta, \delta'))$$

$$\begin{aligned}\mathsf{ic}(x, x', \delta, \delta') \overset{\text{def}}{=}\ & (x{=}0 \wedge x'{=}0) \vee \\ & \Big(x{\neq}0 \wedge \big[(x'{=}0 \wedge \delta^{\mathsf{c}}(x){=}x' \wedge \exists y.\, \delta^{\mathsf{p}}(y){\neq}\varnothing \wedge y \overset{\delta}{\rightsquigarrow}^{\star}_0 x) \\ & \quad \vee \big(x'{\neq}0 \wedge x' \in \delta' \wedge \exists \pi, l, r, l', r'.\, \delta(x){=}((x', \pi, -), l, r) \wedge \delta'(x'){=}(-, l', r') \\ & \qquad \wedge (l'{\neq}0 \Rightarrow \mathsf{ic}(l, l', \delta, \delta')) \wedge (r'{\neq}0 \Rightarrow \mathsf{ic}(r, r', \delta, \delta'))\big) \\ & \quad \vee \big(x'{\neq}0 \wedge x' \in \delta' \wedge \exists l, r, l', r'.\, \delta(x){=}((x', 0, -), l, r) \wedge \delta'(x'){=}(-, l', r') \\ & \qquad \wedge \mathsf{ic}(l, l', \delta, \delta') \wedge \mathsf{ic}(r, r', \delta, \delta')\big)\big]\Big)\end{aligned}$$

$$A^1_\pi \overset{\text{def}}{=} \left\{ \begin{array}{c|l} & \delta_1(x)=\big((0,0,P\uplus\{\pi\}),l,r\big) \land \delta_1{}^{\mathsf{L}}(l)=(c_l,s_l,P_l) \land \delta_1{}^{\mathsf{L}}(r)=(c_r,s_r,P_r) \\ \big((\delta_1,\delta_2), & \land \delta'_1 = (\delta^{\mathsf{V}}_1, \delta^{\mathsf{E}}_1, L'_1) \land \delta'_2 = (V'_2, E'_2, L'_2) \\ (\delta'_1,\delta'_2)\big) & \land L'_1=\delta^{\mathsf{L}}_1[x \mapsto (y,\pi,P)][l \mapsto c_l, s_l, P_l \uplus \{\pi.\mathsf{l}\}][r \mapsto c_r, s_r, P_r\uplus\{\pi.\mathsf{r}\}] \\ & \land V'_2=\delta^{\mathsf{V}}_2\uplus\{y\} \land E'_2=\delta^{\mathsf{E}}_2\uplus[y \mapsto (0,0)] \land L'_2=\delta^{\mathsf{L}}_2\uplus[y \mapsto (0,\pi,\emptyset)]\big) \end{array} \right\}$$

$$A^2_\pi \overset{\text{def}}{=} \left\{ \begin{array}{c|l} \big((\delta_1,\delta_2), & \delta_1(x)=\big((y,\pi,P),l,-\big) \land \big((l=0 \land c_l=0) \lor (\delta^{\mathsf{c}}_1(l)=c_l \land c_l \neq 0)\big) \\ (\delta_1,\delta'_2)\big) & \land\, \delta_2(y)=\big((0,\pi,\emptyset),0,r\big) \land \delta'_2=(\delta^{\mathsf{V}}_2, E'_2, \delta^{\mathsf{L}}_2) \land E'_2=\delta^{\mathsf{E}}_2[y \mapsto (c_l,r)] \end{array} \right\}$$

$$A^3_\pi \overset{\text{def}}{=} \left\{ \begin{array}{c|l} \big((\delta_1,\delta_2), & \delta_1(x)=\big((y,\pi,P),-,r\big) \land \big((r=0 \land c_r=0) \lor (\delta^{\mathsf{c}}_1(r)=c_r \land c_r \neq 0)\big) \\ (\delta_1,\delta'_2)\big) & \land \delta_2(y)=\big((0,\pi,\emptyset),l,0\big) \land \delta'_2=(\delta^{\mathsf{V}}_2, E'_2, \delta^{\mathsf{L}}_2) \land E'_2=\delta^{\mathsf{E}}_2[y \mapsto (l,c_r)] \end{array} \right\}$$

$$A^4_\pi \overset{\text{def}}{=} \left\{ \begin{array}{c|l} \big((\delta_1,\delta_2), & \delta_1(x)=\big((y,\pi,P),l,r\big) \land \delta_2(y)=\big((0,\pi,\emptyset),c_l,c_r\big) \\ (\delta'_1,\delta'_2)\big) & \land (l=0 \land c_l=0 \lor \delta^{\mathsf{c}}_1(l)=c_l \land c_l \neq 0) \land (r=0 \land c_r=0 \lor \delta^{\mathsf{c}}_1(r)=c_r \land c_r \neq 0) \\ & \land\, \delta'_1=(\delta^{\mathsf{V}}_1, \delta^{\mathsf{E}}_1, \delta^{\mathsf{L}}_1[x \mapsto (y,0,P)]) \land \delta'_2=(\delta^{\mathsf{V}}_2, \delta^{\mathsf{E}}_2, \delta^{\mathsf{L}}_2[y \mapsto (0,0,\emptyset)]) \end{array} \right\}$$

$$A^5_\pi \overset{\text{def}}{=} \Big\{\big((\delta_1,\delta_2),(\delta'_1,\delta_2)\big) \,\Big|\, \delta^{\mathsf{L}}_1(x)=(y,s,P\uplus\{\pi\}) \land y \neq 0 \land \delta'_1=\big(\delta^{\mathsf{V}}_1, \delta^{\mathsf{E}}_1, \delta^{\mathsf{L}}_1[x \mapsto (y,s,P)]\big)\Big\}$$

Figure 3: The mathematical actions of `copy_dag`

with $\mathsf{acyc}(\delta) \overset{\text{def}}{=} \neg\exists x.\, x \overset{\delta}{\rightsquigarrow}{}^{+}\, x$, where $\overset{\delta}{\rightsquigarrow}{}^{+}$ denotes the transitive closure of $\overset{\delta}{\rightsquigarrow}$.

Informally, the invariant asserts that δ and δ' are acyclic (first two conjuncts), and that each node x' of the copy dag δ' corresponds to a unique node x of the source dag δ (third conjunct). The last conjunct states that each node x of the source dag (i.e. $x \neq 0$) is in one of the three stages described above, via the second disjunct of the ic predicate: i) x is not copied yet (stage 1), in which case there is an unprocessed path from a node y with a non-empty promise set to x, ensuring that it will eventually be visited (first disjunct); ii) x is currently being processed (stage 2) by thread π (second disjunct), and if its children have been copied they also satisfy the invariant; iii) x has been processed completely (stage 3) and thus its children also satisfy the invariant (last disjunct).

The mathematical precondition of `copy_dag`, $\mathsf{P}^\pi(x,\delta)$, is defined below where x identifies the top node being copied (the argument to `copy_dag`), π denotes the thread identifier, and δ is the source dag. It asserts that π is in the promise set of x, i.e. thread π has an obligation to visit x (first conjunct). Recall that each token uniquely identifies a thread and thus the descendants of π correspond to the sub-threads subsequently spawned by π. As such, prior to spawning new threads the precondition asserts that none of the strict descendants of π can be found anywhere in the promise sets (second conjunct), and π itself is only in the promise set of x (third conjunct). Similarly, neither π nor its descendants have yet processed any nodes (last conjunct). The mathematical postcondition, $\mathsf{Q}^\pi(x,y,\delta,\delta')$, is as defined below and asserts that x (in δ) has been copied to y (in δ'); that π and all its descendants have fulfilled their promises and thus cannot be found in promise sets; and that π and all its descendants have finished processing their charges and thus cannot correspond to the stage field of a node.

$$\begin{aligned} \mathsf{P}^\pi(x,\delta) &\overset{\text{def}}{=} (x=0 \lor \pi \in \delta^{\mathsf{p}}(x)) \land \forall\pi'.\, \forall y \in \delta. \\ &\qquad (\pi' \in \delta^{\mathsf{p}}(y) \Rightarrow \pi' \not\sqsubset \pi) \land (x \neq y \Rightarrow \pi \notin \delta^{\mathsf{p}}(y)) \land (\delta^{\mathsf{s}}(y)=\pi' \Rightarrow \pi' \not\sqsubseteq \pi) \\ \mathsf{Q}^\pi(x,y,\delta,\delta') &\overset{\text{def}}{=} (x=0 \lor (\delta^{\mathsf{c}}(x)=y \land y \in \delta')) \land \forall\pi'.\, \forall z \in \delta. \\ &\qquad \pi' \in \delta^{\mathsf{p}}(z) \lor \delta^{\mathsf{s}}(z)=\pi' \Rightarrow \pi' \not\sqsubseteq \pi \end{aligned}$$

Observe that when the top level thread (associated with $\bullet$) executing `copy_dag(x)` terminates, since $\bullet$ is the maximal token and all other tokens are its descendants (i.e. $\forall \pi.\, \pi \sqsubseteq \bullet$), the second conjunct of $\mathsf{Q}^{\bullet}(\mathtt{x}, \mathtt{ret}, \delta, \delta')$ entails that no tokens can be found anywhere in δ, i.e. $\forall y.\, \delta^{\mathsf{p}}(y)=\emptyset \wedge \delta^{\mathsf{s}}(y)=0$. As such, $\mathsf{Q}^{\bullet}(\mathtt{x}, \mathtt{ret}, \delta, \delta')$ together with Inv entails that all nodes in δ have been correctly copied into δ', i.e. only the third disjunct of $\mathsf{ic}(\mathtt{x}, \mathtt{ret}, \delta, \delta')$ in Inv applies.

Recall from §2.2 that as a key proof obligation we must prove that our mathematical assertions are stable with respect to our mathematical actions. This is captured by Lemma 1 below. Part (1) states that the invariant Inv is stable with respect to the actions of all threads. That is, if the invariant holds for (δ_1, δ_2), and a thread π updates (δ_1, δ_2) to (δ_3, δ_4), then the invariant holds for (δ_3, δ_4). Parts (2) and (3) state that the pre- and postconditions of thread π' ($\mathsf{P}^{\pi'}$ and $\mathsf{Q}^{\pi'}$) are stable with respect to the actions of all threads π, but those of its descendants ($\pi \notin \overline{\pi'}$). Observe that despite this latter stipulation, the actions of π are irrelevant and do not affect the stability of $\mathsf{P}^{\pi'}$ and $\mathsf{Q}^{\pi'}$. More concretely, the precondition $\mathsf{P}^{\pi'}$ only holds at the beginning of the program *before* new descendants are spawned (line 9). As such, at these program points $\mathsf{P}^{\pi'}$ is trivially stable with respect to the actions of its (non-existing) descendants. Analogously, the postcondition $\mathsf{Q}^{\pi'}$ only holds at the end of the program *after* the descendant threads have completed their execution and joined. Therefore, at these program points $\mathsf{Q}^{\pi'}$ is trivially stable with respect to the actions of its descendants.

Lemma 1. *For all mathematical objects* $(\delta_1,\delta_2), (\delta_3,\delta_4)$*, and all tokens* π, π'*,*

$$\mathsf{Inv}(\delta_1, \delta_2) \wedge (\delta_1,\delta_2)\, A_\pi\, (\delta_3,\delta_4) \Rightarrow \mathsf{Inv}(\delta_3, \delta_4) \tag{1}$$
$$\mathsf{P}^{\pi'}(x, \delta_1) \wedge (\delta_1,\delta_2)\, A_\pi\, (\delta_3,\delta_4) \wedge \pi \notin \overline{\pi'} \Rightarrow \mathsf{P}^{\pi'}(x, \delta_3) \tag{2}$$
$$\mathsf{Q}^{\pi'}(x, y, \delta_1, \delta_2) \wedge (\delta_1,\delta_2)\, A_\pi\, (\delta_3,\delta_4) \wedge \pi \notin \overline{\pi'} \Rightarrow \mathsf{Q}^{\pi'}(x, y, \delta_3, \delta_4) \tag{3}$$

Proof. Follows from the definitions of A_π, Inv, P, and Q. The full proof is given in [10].

We are almost in a position to verify `copy_dag`. As discussed in §2.2, in order to verify `copy_dag` we integrate our mathematical correctness argument with a machine-level memory safety argument by linking our abstract mathematical objects to concrete structures in the heap. We proceed with the spatial representation of our mathematical dags in the heap.

Spatial representation We represent a mathematical object (δ, δ') in the heap through the icdag (in-copy) predicate below as two disjoint ($*$-separated) dags, as well as a ghost location (d) in the ghost heap tracking the current abstract state of each dag. Observe that this way of tracking the abstract state of dags in the ghost heap eliminates the need for baking in the abstract state into the model. That is, rather than incorporating the abstract state into the model as in [15, 16], we encode it as an additional resource in the ghost heap. We use $\Rightarrow$ for ghost heap cells to differentiate them from concrete heap cells indicated by $\mapsto$. We implement each dag as a collection of nodes in the heap. A node is represented as three adjacent cells in the heap together with two additional cells in the ghost heap. The cells in the heap track the addresses of the copy (c), and the left (l) and right (r) children, respectively. The ghost locations are used to track the node state (s) and the promise set (P). It is also possible (and perhaps more pleasing) to implement a dag via a *recursive* predicate using the overlapping conjunction $\uplus$ (see [10]). Here, we choose the implementation below for simplicity.

$$\mathsf{icdag}(\delta_1, \delta_2) \overset{\text{def}}{=} d \Rightarrow (\delta_1, \delta_2) * \mathsf{dag}(\delta_1) * \mathsf{dag}(\delta_2) \qquad \mathsf{dag}(\delta) \overset{\text{def}}{=} \mathop{\bigstar}_{x \in \delta} \mathsf{node}(x, \delta)$$
$$\mathsf{node}(x, \delta) \overset{\text{def}}{=} \exists l, r, c, s, P.\, \delta(x) = (c, s, P), l, r \wedge x \mapsto c, l, r * x \Rightarrow s, P$$

We can now specify the spatial precondition of `copy_dag`, $\mathsf{Pre}(x, \pi, \delta)$, as a CoLoSL assertion defined below where x is the top node being copied (the argument of `copy_dag`), π identifies the running thread, and δ denotes the initial top-level dag (where none of the nodes are copied yet). Recall that the spatial actions in CoLoSL are indexed by *capabilities*; that is, a CoLoSL action may be performed by a thread only when it holds the necessary capabilities. Since CoLoSL is parametric in its capability model, to verify `copy_dag` we take our capabilities to be the same as our tokens. The precondition Pre states that the current thread π holds the capabilities associated with itself and all its descendants ($\overline{\pi}^*$). Thread π will subsequently pass on the descendant capabilities when spawning new sub-threads and reclaim them as the sub-threads return and join. The Pre further asserts that the initial dag δ and its copy currently correspond to δ_1 and δ_2, respectively. That is, since the dags are concurrently manipulated by several threads, to ensure the stability of the shared state assertion to the actions of the environment, Pre states that the initial dag δ may have evolved to another congruent dag δ_1 (captured by the existential quantifier). The Pre also states that the shared state contains the spatial resources of the dags ($\mathsf{icdag}(\delta_1, \delta_2)$), that (δ_1, δ_2) satisfies the invariant Inv, and that the source dag δ_1 satisfies the mathematical precondition P^π. The spatial actions on the shared state are declared in I where mathematical actions are simply lifted to spatial ones indexed by the associated capability. That is, if thread π holds the π capability, and the actions of π (A_π) admit the update of the mathematical object (δ_1, δ_2) to (δ_1', δ_2'), then thread π may update the spatial resources $\mathsf{icdag}(\delta_1, \delta_2)$ to $\mathsf{icdag}(\delta_1', \delta_2')$. Finally, the spatial postcondition Post is analogous to Pre and further states that node x has been copied to y.

$$\mathsf{Pre}(x, \pi, \delta) \overset{\text{def}}{=} \overline{\pi}^* * \boxed{\exists \delta_1, \delta_2.\, \mathsf{icdag}(\delta_1, \delta_2) * (\delta \dot{\cong} \delta_1 \wedge \mathsf{Inv}(\delta_1, \delta_2) \wedge \mathsf{P}^\pi(x, \delta_1))}_I$$
$$\mathsf{Post}(x, y, \pi, \delta) \overset{\text{def}}{=} \overline{\pi}^* * \boxed{\exists \delta_1, \delta_2.\, \mathsf{icdag}(\delta_1, \delta_2) * (\delta \dot{\cong} \delta_1 \wedge \mathsf{Inv}(\delta_1, \delta_2) \wedge \mathsf{Q}^\pi(x, y, \delta_1, \delta_2))}_I$$
$$\overline{\pi}^* \overset{\text{def}}{=} \mathop{\bigstar}_{\pi \in \overline{\pi}} \pi \qquad I \overset{\text{def}}{=} \left\{ \pi : \mathsf{icdag}(\delta_1, \delta_2) \wedge (\delta_1, \delta_2) A_\pi (\delta_1', \delta_2') \rightsquigarrow \mathsf{icdag}(\delta_1', \delta_2') \right.$$

Verifying `copy_dag` We give a proof sketch of `copy_dag` in Fig. 4. At each proof point, we have highlighted the effect of the preceding command, where applicable. For instance, after line 4 we allocate a new node in the heap at `y` as well as two consecutive cells in the ghost heap at `y`. One thing jumps out when looking at the assertions at each program point: they have *identical* spatial

1. `struct node {struct node *c, *l, *r};`
   $\{\mathsf{Pre}(\mathsf{x},\pi,\delta)\}$
2. `copy_dag(struct node *x) {struct node *l,*r,*ll,*rr,*y; bool b;`
   $\left\{\overline{\pi}^* * \boxed{\exists\delta_1,\delta_2.\,\mathsf{icdag}(\delta_1,\delta_2) * (\delta\dot{\cong}\delta_1 \land \mathsf{Inv}(\delta_1,\delta_2) \land \mathsf{P}^\pi(x,\delta_1))}_I\right\}$
3. `  if(!x){ return 0; }`
   $\left\{\overline{\pi}^* * \mathsf{ret}\doteq 0 * \boxed{\exists\delta_1,\delta_2.\mathsf{icdag}(\delta_1,\delta_2) * (\delta\dot{\cong}\delta_1 \land \mathsf{Inv}(\delta_1,\delta_2) \land \mathsf{Q}^\pi(x,\mathsf{ret},\delta_1,\delta_2))}_I\right\}$
4. `  y = malloc(sizeof(struct node));`
   $\left\{\overline{\pi}^* * \boxed{\exists\delta_1,\delta_2.\,\mathsf{icdag}(\delta_1,\delta_2) * (\delta\dot{\cong}\delta_1 \land \mathsf{Inv}(\delta_1,\delta_2) \land \mathsf{P}^\pi(x,\delta_1))}_I * \mathsf{y}\mapsto 0,0,0 * \mathsf{y}\Rightarrow\pi,\varnothing\right\}$
5. `  <if(x->c){ b = false;` $//$Perform the action A_π^5
   $\left\{\begin{array}{l}\overline{\pi}^* * \boxed{\exists\delta_1,\delta_2.\,\mathsf{icdag}(\delta_1,\delta_2) * (\delta\dot{\cong}\delta_1 \land \mathsf{Inv}(\delta_1,\delta_2) \land \mathsf{Q}^\pi(x,\delta_1^{\mathsf{c}}(\mathsf{x}),\delta_1,\delta_2) \land \delta_1^{\mathsf{c}}(\mathsf{x}) \neq 0)}_I \\ * \mathsf{y}\mapsto 0,-,- * \mathsf{y}\Rightarrow\pi,\varnothing * \mathsf{b}\doteq 0\end{array}\right\}$
6. `  }else{ x->c = y; b = true;` $//$Perform the action A_π^1
   $\left\{\overline{\pi}^* * \boxed{\begin{array}{l}\exists\delta_1,\delta_2.\,\mathsf{icdag}(\delta_1,\delta_2) * (\delta\dot{\cong}\delta_1 \land \mathsf{Inv}(\delta_1,\delta_2) \land \forall y{\in}\delta_1.\,\pi\notin\delta_1^{\mathsf{p}}(y) \land \\ (\mathsf{x}\neq y \Rightarrow \pi\neq\delta_1^{\mathsf{s}}(y)) \land \exists l,r.\,\delta_1(\mathsf{x})=(\mathsf{y},\pi,-,l,r) \land \mathsf{y}\dot{\in}\delta_2 \land \mathsf{P}^{\pi.\mathsf{l}}(l,\delta_1) \land \mathsf{P}^{\pi.\mathsf{r}}(r,\delta_1))\end{array}}_I * \mathsf{b}\doteq 1\right\}$
7. `  }>`
8. `  if(b){ l = x->l; r = x->r;`
   $\left\{\overline{\pi}^* * \boxed{\begin{array}{l}\exists\delta_1,\delta_2.\,\mathsf{icdag}(\delta_1,\delta_2) * (\delta\dot{\cong}\delta_1 \land \mathsf{Inv}(\delta_1,\delta_2) \land \forall y{\in}\delta_1.\,\pi\notin\delta_1^{\mathsf{p}}(y) \land \\ (\mathsf{x}\neq y \Rightarrow \pi\neq\delta_1^{\mathsf{s}}(y)) \land \delta_1(\mathsf{x})=(\mathsf{y},\pi,-,\mathtt{l},\mathtt{r}) \land \mathsf{y}\dot{\in}\delta_2 \land \mathsf{P}^{\pi.\mathsf{l}}(\mathtt{l},\delta_1) \land \mathsf{P}^{\pi.\mathsf{r}}(\mathtt{r},\delta_1))\end{array}}_I\right\}$
   $\left\{\begin{array}{l}\pi * \boxed{\begin{array}{l}\exists\delta_1,\delta_2.\mathsf{icdag}(\delta_1,\delta_2) * (\delta\dot{\cong}\delta_1 \land \mathsf{Inv}(\delta_1,\delta_2) \land \forall y{\in}\delta_1.\,\pi\notin\delta_1^{\mathsf{p}}(y) \land \\ (\mathsf{x}\neq y \Rightarrow \pi\neq\delta_1^{\mathsf{s}}(y)) \land \delta_1(\mathsf{x})=(\mathsf{y},-,\pi,\mathtt{l},\mathtt{r}) \land \mathsf{y}\dot{\in}\delta_2)\end{array}}_I \\ * \overline{\pi.\mathsf{l}}^* * \boxed{\exists\delta_1,\delta_2.\,\mathsf{icdag}(\delta_1,\delta_2) * (\delta\dot{\cong}\delta_1 \land \mathsf{Inv}(\delta_1,\delta_2) \land \mathsf{P}^{\pi.\mathsf{l}}(\mathtt{l},\delta_1))}_I \\ * \overline{\pi.\mathsf{r}}^* * \boxed{\exists\delta_1,\delta_2.\,\mathsf{icdag}(\delta_1,\delta_2) * (\delta\dot{\cong}\delta_1 \land \mathsf{Inv}(\delta_1,\delta_2) \land \mathsf{P}^{\pi.\mathsf{r}}(\mathtt{r},\delta_1))}_I\end{array}\right\}$
   $\left\{\pi * \boxed{\begin{array}{l}\exists\delta_1,\delta_2.\mathsf{icdag}(\delta_1,\delta_2) * (\delta\dot{\cong}\delta_1 \land \mathsf{Inv}(\delta_1,\delta_2) \land \forall y{\in}\delta_1.\,\pi\notin\delta_1^{\mathsf{p}}(y) \\ \land\ (\mathsf{x}\neq y \Rightarrow \pi\neq\delta_1^{\mathsf{s}}(y)) \land \delta_1(\mathsf{x})=(\mathsf{y},-,\pi,\mathtt{l},\mathtt{r}) \land \mathsf{y}\dot{\in}\delta_2)\end{array}}_I \begin{array}{l}* \mathsf{Pre}(\mathtt{l},\pi.\mathsf{l},\delta) \\ * \mathsf{Pre}(\mathtt{r},\pi.\mathsf{r},\delta)\end{array}\right\}$
9. $\{\mathsf{Pre}(\mathtt{l},\pi.\mathsf{l},\delta)\}$ `ll = copy_dag(l)` $\{\mathsf{Post}(\mathtt{l},\mathtt{ll},\pi.\mathsf{l},\delta)\}$ $\|$ $\{\mathsf{Pre}(\mathtt{r},\pi.\mathsf{r},\delta)\}$ `rr = copy_dag(r)` $\{\mathsf{Post}(\mathtt{r},\mathtt{rr},\pi.\mathsf{r},\delta)\}$
   $\left\{\pi * \boxed{\begin{array}{l}\exists\delta_1,\delta_2.\mathsf{icdag}(\delta_1,\delta_2) * (\delta\dot{\cong}\delta_1 \land \mathsf{Inv}(\delta_1,\delta_2) \land \forall y{\in}\delta_1.\,\pi\notin\delta_1^{\mathsf{p}}(y) \\ \land\ (\mathsf{x}\neq y \Rightarrow \pi\neq\delta_1^{\mathsf{s}}(y)) \land \delta_1(\mathsf{x})=(\mathsf{y},-,\pi,\mathtt{l},\mathtt{r}) \land \mathsf{y}\dot{\in}\delta_2)\end{array}}_I \begin{array}{l}* \mathsf{Post}(\mathtt{l},\mathtt{ll},\pi.\mathsf{l},\delta) \\ * \mathsf{Post}(\mathtt{r},\mathtt{rr},\pi.\mathsf{r},\delta)\end{array}\right\}$
   $\left\{\overline{\pi}^* * \boxed{\begin{array}{l}\exists\delta_1,\delta_2.\mathsf{icdag}(\delta_1,\delta_2) * (\delta\dot{\cong}\delta_1 \land \mathsf{Inv}(\delta_1,\delta_2) \land \forall y{\in}\delta_1.\pi\notin\delta_1^{\mathsf{p}}(y)\ \land \\ (\mathsf{x}\neq y \Rightarrow \pi\neq\delta_1^{\mathsf{s}}(y)) \land \delta_1(\mathsf{x})=(\mathsf{y},-,\pi,\mathtt{l},\mathtt{r}) \land \mathsf{y}\dot{\in}\delta_2 \land \mathsf{Q}^{\pi.\mathsf{l}}(\mathtt{l},\mathtt{ll},\delta_1,\delta_2) \land \mathsf{Q}^{\pi.\mathsf{r}}(\mathtt{r},\mathtt{rr},\delta_1,\delta_2))\end{array}}_I\right\}$
10. `    <y->l = ll>; <y->r = rr>;` $//$Perform A_π^2, A_π^3 and A_π^4 in order.
    $\left\{\overline{\pi}^* * \boxed{\exists\delta_1,\delta_2.\,\mathsf{icdag}(\delta_1,\delta_2) * (\delta\dot{\cong}\delta_1 \land \mathsf{Inv}(\delta_1,\delta_2) \land \mathsf{Q}^\pi(\mathsf{x},\mathsf{y},\delta_1,\delta_2))}_I\right\}$
11. `    return y;` $\left\{\overline{\pi}^* * \boxed{\exists\delta_1,\delta_2.\,\mathsf{icdag}(\delta_1,\delta_2) * (\delta\dot{\cong}\delta_1 \land \mathsf{Inv}(\delta_1,\delta_2) \land \mathsf{Q}^\pi(\mathsf{x},\mathsf{ret},\delta_1,\delta_2))}_I\right\}$
12. `  }else{`
    $\left\{\overline{\pi}^* * \boxed{\exists\delta_1,\delta_2.\,\mathsf{icdag}(\delta_1,\delta_2) * (\delta\dot{\cong}\delta_1 \land \mathsf{Inv}(\delta_1,\delta_2) \land \mathsf{Q}^\pi(\mathsf{x},\delta_1^{\mathsf{c}}(\mathsf{x}),\delta_1,\delta_2) \land \delta_1^{\mathsf{c}}(\mathsf{x})\dot{\neq}0)}_I \begin{array}{l}* \mathsf{y}\mapsto 0,-,- \\ * \mathsf{y}\Rightarrow\pi,\varnothing\end{array}\right\}$
13. `    free(y, sizeof(struct node)) ; return x->c;`
    $\left\{\overline{\pi}^* * \boxed{\exists\delta_1,\delta_2.\,\mathsf{icdag}(\delta_1,\delta_2) * (\delta\dot{\cong}\delta_1 \land \mathsf{Inv}(\delta_1,\delta_2) \land \mathsf{Q}^\pi(\mathsf{x},\mathsf{ret},\delta_1,\delta_2))}_I\right\}$
14. `} }` $\{\mathsf{Post}(\mathsf{x},\mathsf{ret},\pi,\delta)\}$

Figure 4: The code and a proof sketch of `copy_dag`

parts in the shared state: $\mathsf{icdag}(\delta_1, \delta_2)$. Indeed, the spatial graph in the heap is changing constantly, due both to the actions of this thread and the environment. Nevertheless, the spatial graph in the heap remains in sync with the mathematical object (δ_1, δ_2), however (δ_1, δ_2) may be changing. Whenever this thread interacts with the shared state, the mathematical object (δ_1, δ_2) changes, reflected by the changes to the pure mathematical facts. Changes to (δ_1, δ_2) due to other threads in the environment are handled by the existential quantification of δ_1 and δ_2.

On line 3 we check if x is 0. If so the program returns and the postcondition, $\mathsf{Post}(\mathsf{x}, 0, \delta, \pi)$, follows trivially from the definition of the precondition $\mathsf{Pre}(\mathsf{x}, \delta, \pi)$. If $\mathsf{x} \neq 0$, then the atomic block of lines 5-7 is executed. We first check if x is copied; if so we set b to false, perform action A^5_π (i.e. remove π from the promise set of x) and thus arrive at the desired postcondition $\mathsf{Post}(\mathsf{x}, \delta^{\mathsf{c}}_1(\mathsf{x}), \pi, \delta)$. On the other hand, if x is not copied, we set b to true and perform A^1_π. That is, we remove π from the promise set of x, and add $\pi.\mathsf{l}$ and $\pi.\mathsf{r}$ to the left and right children of x, respectively. In doing so, we obtain the mathematical preconditions $\mathsf{P}^{\delta_1}(l, \pi.\mathsf{l})$ and $\mathsf{P}^{\delta_1}(r, \pi.\mathsf{r})$. On line 8 we check whether the thread did copy x and has thus incurred an obligation to call copy_dag on x's children. If this is the case, we load the left and right children of x into l and r, and subsequently call copy_dag on them (line 9). To obtain the preconditions of the recursive calls, we duplicate the shared state twice ($\boxed{P}_I \overset{\text{Copy}\times 2}{\Longrightarrow} \boxed{P}_I * \boxed{P}_I * \boxed{P}_I$), drop the irrelevant pure assertions, and unwrap the definition of $\overline{\pi}^{*}$. We then use the Par rule (Fig. 1) to distribute the resources between the sub-threads and collect them back when they join. Subsequently, we combine multiple copies of the shared states into one using Merge. Finally, on line 10 we perform actions A^2_π, A^3_π and A^4_π in order to update the edges of y, and arrive at the postcondition $\mathsf{Post}(\mathsf{x}, \mathsf{y}, \pi, \delta)$.

Copying graphs Recall that a dag is a directed graph that is *acyclic*. However, the copy_dag program does not depend on the acyclicity of the dag at x and thus copy_dag may be used to copy *both* dags and cyclic graphs. The specification of copy_dag for cyclic graphs is rather similar to that of dags. More concretely, the spatial pre- and postcondition (Pre and Post), as well as the mathematical pre- and postcondition (P and Q) remain unchanged, while the invariant Inv is weakened to allow for cyclic graphs. That is, the Inv for cyclic graphs does not include the first two conjuncts asserting that δ and δ' are acyclic. As such, when verifying copy_dag for cyclic graphs, the proof obligation for establishing the Inv stability (i.e. Lemma 1(1)) is somewhat simpler. The other stability proofs (Lemma 1(2) and (3)) and the proof sketch in Fig. 4 are essentially unchanged.

4 Parallel Speculative Shortest Path (Dijkstra)

Given a graph with size vertices, the weighted adjacency matrix a, and a designated source node src, Dijkstra's sequential algorithm calculates the shortest path from src to all other nodes incrementally. To do this, it maintains a cost array c, and two sets of vertices: those processed thus far (done), and those yet to be processed (work). The cost for each node (bar src itself) is initialised with the value of the adjacency matrix (i.e. $\mathsf{c[src]}{=}0$; $\mathsf{c[i]}{=}\mathsf{a[src][i]}$ for $\mathsf{i} \neq \mathsf{src}$).

Initially, all vertices are in `work` and the algorithm proceeds by iterating over `work` performing the following two steps at each iteration. First, it extracts a node `i` with the *cheapest* cost from `work` and inserts it to `done`. Second, for each vertex `j`, it updates its cost (`c[j]`) to $min\{$`c[j]`, `c[i]`+`a[i][j]`$\}$. This greedy strategy ensures that at any one point the cost associated with the nodes in `done` is minimal. Once the `work` set is exhausted, `c` holds the minimal cost for all vertices.

We study a *parallel non-greedy* variant of Dijkstra's shortest path algorithm, `parallel_dijkstra` in Fig. 5, with `work` and `done` implemented as bit arrays. We initialize the `c`, `work` and `done` arrays as described above (lines 2-5), and find the shortest path from the source `src` concurrently, by spawning multiple threads, each executing the non-greedy `dijkstra` (line 6). The code for `dijkstra` is given in Fig. 5. In this non-greedy implementation, at each iteration an *arbitrary* node from the `work` set is selected rather than one with minimal cost. Unlike the greedy variant, when a node is processed and inserted into `done`, its associated cost is not necessarily the cheapest. As such, during the second step of each iteration, when updating the cost of node `j` to $min\{$`c[j]`,`c[i]`+`a[i][j]`$\}$ (as described above), we must further check if `j` is already processed. This is because if the cost of `j` goes down, the cost of its adjacent siblings may go down too and thus `j` needs to be *reprocessed*. When this is the case, `j` is removed from `done` and reinserted into `work` (lines 9-11). If on the other hand `j` is unprocessed (and is in `work`), we can safely decrease its cost (lines 7-8). Lastly, if `j` is currently being processed by another thread, we must wait until it is processed (loop back and try again).

The algorithm of `parallel_dijkstra` is an instance of *speculative parallelism* [7]: each thread running `dijkstra` assumes that the costs of the nodes in `done` will not change as a result of processing the nodes in `work` and proceeds with its computation. However, if at a later point it detects that its assumption was wrong, it reinserts the affected nodes into `work` and recomputes their costs.

Mathematical graphs Similar to dags in §3, we define our mathematical graphs, $\gamma \in \Gamma$, as tuples of the form (V, E, L) where V is the set of vertices, $E : V \to (V \to \mathcal{W})$ is the weighted adjacency function with weights $\mathcal{W} \overset{\text{def}}{=} \mathbb{N} \uplus \{\infty\}$, and $L : V \to D$ is the label function, with the labels D defined shortly. We use the matrix notation for adjacency functions and write $E[i][j]$ for $E(i)(j)$.

Unlike `copy_dag` in §3 where a new thread is spawned at every recursive call point, in `parallel_dijkstra` the number of threads to run concurrently is decided at the beginning (line 7) and remains unchanged thereafter. This allows for a simpler token mechanism. We define our tokens as elements of the (countably) infinite set $t \in \Theta \overset{\text{def}}{=} \mathbb{N} \backslash \{0, 1\}$. We refer to the thread with token t simply as thread t. Recall that each node x in the graph can be either: unprocessed (in `work`); processed (in `done`); or under process by a thread (neither in `work` nor in `done`). We define our labels as $D \overset{\text{def}}{=} \mathcal{W} \times \big(\{0, 1\} \uplus \Theta\big) \times (V \to \{\circ, \bullet\} \uplus \mathcal{W})$. The first component denotes the cost of the shortest path from the source (so far) to the node. The second component describes the node state (0 for unprocessed, 1 for processed, and t when under process by thread t). The last component denotes the *responsibility* function. Recall that when a thread is processing a node, it iterates over all vertices examining whether their cost can be improved.

```
1  void parallel_dijkstra(int[][] a, int[] c, int size, src){
2   bitarray work[size], done[size];
3    for (i=0; i<size; i++){
4      c[i] = a[src][i]; work[i] = 1; done[i] = 0;
5    }; c[src] = 0;
6    dijkstra(a,c,size,work,done) || ... || dijkstra(a,c,size,work,done)
7    return c;
8  }
```

```
1  void dijkstra(int[][] a, int[] c, int size, bitarray work, done){ i = 0;
2    while(done != 2^size-1){ b = <CAS(work[i], 1, 0)>;
3      if(b){ cost = c[i];
4        for(j=0; j<size; j++){ newcost = cost + a[i][j];  b = true;
5          do{ oldcost = c[j];
6            if(newcost < oldcost){
7              b = <CAS(work[j], 1, 0)>;
8              if(b){ b = <CAS(c[j], oldcost, newcost)>; <work[j] = 1>; }
9              else { b = <CAS(done[j], 1, 0)>;
10               if(b){ b = <CAS(c[j], oldcost, newcost)>;
11                 if(b){ < work[j] = 1 > } else { < done[j] = 1 > }
12           } } }
13         } while(!b)
14       } < done[i] = 1 >;
15     } i = (i+1) mod size;
16 } }
```

Figure 5: A parallel non-greedy variant of Dijkstra's algorithm

To do this, at each iteration the thread records the current cost of node j under inspection in `oldcost` (line 5). If the cost may be improved (i.e. the conditional of line 6 succeeds), it then *attempts* to update the cost of j with the improved value (lines 8, 10). Note that since the cost associated with j may have changed from the initial cost recorded (`oldcost`), the update operation may fail and thus the thread needs to re-examine j. To track the iteration progress, for each node the responsibility function records whether i) its cost is yet to be examined ($\circ$); ii) its cost has been examined ($\bullet$); or iii) its cost is currently being examined ($c \in \mathcal{W}$) with its initial cost recorded as c (`oldcost`$=c$). We use the string notation for responsibility functions and write e.g. $\bullet^n.c.\circ^m$, when the first n nodes are mapped to $\bullet$, the (n+1)st node is mapped to c, and the last m nodes are mapped to $\circ$. We write $\bigcirc$ (resp. ⬤) for a function that maps all elements to $\circ$ (resp. $\bullet$).

Given a graph $\gamma{=}(V, E, L)$, we write γ^{V} for V, γ^{E} for E, and γ^{L} for L. We write $\gamma^{\mathsf{c}}(x)$, $\gamma^{\mathsf{s}}(x)$ and $\gamma^{\mathsf{r}}(x)$, for the first, second and third projections of $L(x)$, respectively. Two graphs are *congruent* if they have equal vertices and edges: $\gamma_1 \cong \gamma_2 \overset{\text{def}}{=} \gamma_1^{\mathrm{V}}{=}\gamma_2^{\mathrm{V}} \wedge \gamma_1^{\mathrm{E}}{=}\gamma_2^{\mathrm{E}}$. We define the weighted path relation ($\overset{\gamma}{\rightsquigarrow}_c$), and its reflexive transitive closure as:

$$x \overset{\gamma}{\rightsquigarrow}_c y \overset{\text{def}}{=} (\gamma^{\mathrm{E}})[x][y]{=}c \qquad x \overset{\gamma}{\rightsquigarrow}{}^{*}_{c} y \overset{\text{def}}{=} (x{=}y \wedge c{=}0) \vee (\exists c_1,c_2,z.\, c{=}c_1{+}c_2 \wedge x \overset{\gamma}{\rightsquigarrow}_{c_1} z \wedge z \overset{\gamma}{\rightsquigarrow}{}^{*}_{c_2} y)$$

$$A_t^1 \overset{\text{def}}{=} \big\{\big((V,E,L),(V,E,L')\big) \mid L(i){=}(c,0,\bigcirc) \wedge L'{=}L[i \mapsto (c,t,\bigcirc)]\big\}$$

$$A_t^2 \overset{\text{def}}{=} \left\{\big((V,E,L),(V,E,L')\big) \,\middle|\, \begin{array}{l} L(i){=}(c,t,R) \wedge \forall k{<}j.\, R[k]{=}\bullet \wedge R[j]{=}\circ \\ \wedge\, L(j){=}(c',-,-) \wedge R'{=}R[j \mapsto c'] \wedge L'{=}L[i \mapsto (c,t,R')] \end{array}\right\}$$

$$A_t^3 \overset{\text{def}}{=} \left\{\big((V,E,L),(V,E,L')\big) \,\middle|\, \begin{array}{l} L(i){=}(-,t,R) \wedge R[j]{=}c' \wedge c{+}E[i][j] \leqslant c' \\ \wedge\, L(j){=}(c,s,R') \wedge s{\in}\{0,1\} \wedge L'{=}L[j{\mapsto}(c,t,R')] \end{array}\right\}$$

$$A_t^4 \overset{\text{def}}{=} \left\{\big((V,E,L),(V,E,L')\big) \,\middle|\, \begin{array}{l} L(i){=}(c,t,R) \wedge R[j]{=}c' \wedge L(j){=}(c',t,R'') \\ \wedge\, c''{=}c{+}E[i][j] \wedge c''{<}c' \\ \wedge\, R'{=}R[j \mapsto \bullet] \wedge L'{=}L[i \mapsto (c,t,R')][j \mapsto (c'',t,R'')] \end{array}\right\}$$

$$A_t^5 \overset{\text{def}}{=} \left\{\big((V,E,L),(V,E,L')\big) \,\middle|\, \begin{array}{l} L(i){=}(c,t,R) \wedge R[j]{=}\bullet \wedge L(j){=}(c',t,-) \\ \wedge\, L'{=}L[j \mapsto (c',0,\bigcirc)] \end{array}\right\}$$

$$A_t^6 \overset{\text{def}}{=} \left\{\big((V,E,L),(V,E,L')\big) \,\middle|\, \begin{array}{l} L(i){=}(c,t,R) \wedge R[j]{=}c'' \wedge L(j){=}(c',t,\bigcirc) \wedge c'{\neq}c'' \\ \wedge\, R'{=}R[j{\mapsto}\circ] \wedge L'{=}L[i \mapsto (c,t,R')][j \mapsto (c',0,\bigcirc)] \end{array}\right\}$$

$$A_t^7 \overset{\text{def}}{=} \left\{\big((V,E,L),(V,E,L')\big) \,\middle|\, \begin{array}{l} L(i){=}(c,t,R) \wedge R[j]{=}c'' \wedge L(j){=}(c',t,\text{●}) \wedge c'{\neq}c'' \\ \wedge\, R'{=}R[j \mapsto \circ] \wedge L'{=}L[i \mapsto (c,t,R')][j \mapsto (c',1,\text{●})] \end{array}\right\}$$

$$A_t^8 \overset{\text{def}}{=} \left\{\big((V,E,L),(V,E,L')\big) \,\middle|\, \begin{array}{l} L(i){=}(c,t,R) \wedge R[j]{=}c' \wedge c{+}E[i][j] \geqslant c' \\ \wedge\, R'{=}[j \mapsto \bullet]R \wedge L'{=}[i \mapsto (c,t,R')]L \end{array}\right\}$$

$$A_t^9 \overset{\text{def}}{=} \big\{\big((V,E,L),(V,E,L')\big) \mid L(x){=}(c,t,\text{●}) \wedge L'{=}L[x \mapsto (c,1,\text{●})]\big\}$$

Figure 6: The mathematical actions of `dijkstra`

Actions We define several families of actions in Fig. 6, each of which indexed by a token t. The A_t^1 describes the `CAS` operation of line 2 in the algorithm: the state of a node is changed from unprocessed to being processed by thread t (i is removed from `work`). The A_t^2 describes a *ghost* action at line 5 for iteration j when storing the current cost of j in `oldcost`. The thread has not yet examined the cost of node j ($R[j]{=}\circ$). It then reads the current cost (c') of j and (ghostly) updates the responsibility function. The A_t^3 describes the `CAS` operations of lines 7 and 9 when successful: when processing i, we discovered that the cost of j may be improved ($c{+}E[i][j] \leqslant c'$). In the former case, j is currently unprocessed (in `work`, $s{=}0$), while in the latter j is processed (in `done`, $s{=}1$). In both cases, we remove j from the respective set and temporarily change its state to under process by t until its cost is updated and it is reinserted into the relevant set. The A_t^4 describes the `CAS` operations in lines 8 and 10 when successful. The cost of j has not changed since we first read it ($R[j]{=}c'$) and we discovered that this cost may be improved ($c''{\leqslant}c'$). The responsibility of i towards j is then marked as fulfilled ($R'[j]{=}\bullet$) and the cost of j is updated until it is subsequently reinserted into `work` via A_t^5. The A_t^5 denotes the reinsertion of j into `work` in lines 8 and 11 following *successful* `CAS` operations at lines 8 and 10. The state of j is changed to 0 to reflect its insertion to `work`. The A_t^6 and A_t^7 sets respectively describe the reinsertion of j into `work` and `done` in lines 8 and 11, following *failed* `CAS` operations at lines 8 and 10. When attempting to update the cost of j, we discovered that the cost of j has changed since we first read it ($c'{\neq}c''$). We thus reinsert j into the relevant set and (ghostly) update the responsibility function to reflect that j is to be re-examined ($R'[j]{=}\circ$). The A_t^8 describes a ghost action in line 6 when the conditional fails: examining j yielded no cost improvement and thus the responsibility of i towards j is marked as fulfilled. Lastly, the A_t^9 captures the atomic operation in line 14: processing of i is at an end since all nodes have been examined. The state of i is thus changed to processed (i is inserted into done). We write A_t for actions of t, i.e. $A_t \overset{\text{def}}{=} \bigcup_{i \in \{1 \ldots 9\}} A_t^i$.

Mathematical invariant Throughout the execution of `dijkstra` for a source node src, the graph γ satisfies the invariant $\mathsf{Inv}(src, \gamma)$ described below.

$$\begin{aligned}
\mathsf{Inv}(\gamma, src) \overset{\text{def}}{=} & \forall x \in \gamma.\ \mathsf{min}_\gamma^{src}(x, \gamma^{\mathsf{c}}(x)) \\
& \lor \big(\exists y, z, c.\ \mathsf{min}_\gamma^{src}(y, \gamma^{\mathsf{c}}(y)) \land \gamma(y) {\neq} 1 \land \gamma^{\mathsf{r}}[y][z]{=}0 \\
& \qquad \land y \overset{\gamma}{\rightsquigarrow}_c z \land \mathsf{wit}_\gamma^{src}(\gamma^{\mathsf{c}}(y){+}c, z, x)\big) \\
\mathsf{min}_\gamma^{src}(x, c) \overset{\text{def}}{=} & min\{c' \mid s \overset{\gamma}{\rightsquigarrow}{}^{*}_{c'} x\} = c \\
\mathsf{wit}_\gamma^{src}(c, z, x) \overset{\text{def}}{=} & \mathsf{min}_\gamma^{src}(z, c) \land \gamma^{\mathsf{c}}(z) > c \\
& \land (z{=}x \lor (\exists c', w.\ z \overset{\gamma}{\rightsquigarrow}_{c'} w \land \mathsf{wit}_\gamma^{src}(c{+}c', w, x)))
\end{aligned}$$

The $\mathsf{Inv}(\gamma, src)$ asserts that for any node x, either its associated cost from src is minimal; or there is a minimal path to x from a node y (via z), such that the cost of y is minimal and y is either unprocessed or is being processed. Moreover, none of the nodes along this path (except y) are yet associated with their correct (minimal) cost. As such, when y is finally processed, its effect will be propagated down this path, correcting the costs of the nodes along the way. Observe that when `dijkstra` terminates, since all nodes are processed (i.e. $\forall x.\, \gamma^{\mathsf{s}}(x){=}1$), the $\mathsf{Inv}(\gamma, src)$ entails that the cost associated with all nodes is minimal.

Lemma 2. *For all mathematical graphs γ, γ', source nodes src, and tokens t, the $\mathsf{Inv}(\gamma, src)$ invariant is stable with respect to A_t:*

$$\mathsf{Inv}(\gamma, src) \land \gamma\, A_t\, \gamma' \Rightarrow \mathsf{Inv}(\gamma', src)$$

Proof. Follows from the definitions of A_t and Inv. The full proof is given in [10].

Spatial representation Using the $\mathsf{g}(\gamma)$ predicate below, we represent a mathematical graph γ in the heap as multiple $*$-separated arrays: two bit-arrays for the work and done sets, a two-dimensional array for the adjacency matrix, a one dimensional array for the cost function, and finally two ghost arrays for the label function (one for the responsibility function, another for the node states).

$$\begin{aligned}
\mathsf{g}(\gamma) &\overset{\text{def}}{=} \mathsf{work}(\gamma) * \mathsf{done}(\gamma) * \mathsf{adj}(\gamma) * \mathsf{cost}(\gamma) * \mathsf{resp}(\gamma) * \mathsf{state}(\gamma) \\
\mathsf{work}(\gamma) &\overset{\text{def}}{=} \mathop{\bigstar}_{i \in \{i \mid \gamma^{\mathsf{s}}(i)=0\}} \big(\mathtt{work}[i] \mapsto 1\big) * \mathop{\bigstar}_{i \in \{i \mid \gamma^{\mathsf{s}}(i) \neq 0\}} \big(\mathtt{work}[i] \mapsto 0\big) \\
\mathsf{done}(\gamma) &\overset{\text{def}}{=} \mathop{\bigstar}_{i \in \{i \mid \gamma^{\mathsf{s}}(i)=1\}} \big(\mathtt{done}[i] \mapsto 1\big) * \mathop{\bigstar}_{i \in \{i \mid \gamma^{\mathsf{s}}(i) \neq 1\}} \big(\mathtt{done}[i] \mapsto 0\big) \\
\mathsf{adj}(\gamma) &\overset{\text{def}}{=} \mathop{\bigstar}_{i \in \gamma} \big(\mathop{\bigstar}_{j \in \gamma} \mathsf{a}[i][j] \mapsto \gamma^{\mathrm{E}}[i][j]\big) \qquad \mathsf{cost}(\gamma) \overset{\text{def}}{=} \mathop{\bigstar}_{i \in \gamma} \big(\mathsf{c}[i] \mapsto \gamma^{\mathsf{c}}(i)\big)
\end{aligned}$$

```
$\{\mathsf{Pre}(t,\gamma_0)\}$
1. void dijkstra(int[][] a, int[] c, int size, bitarray work, done){ i=0;
2.  while(done != $2^{\texttt{size}}$-1){ b=<CAS(work[i],1,0)>;  //perform $A_t^1$ if possible
3.   if(b){ $\left\{t * \boxed{\exists\gamma.\, \mathsf{g}(\gamma) * (\gamma_0 \dot{\cong} \gamma \land \mathsf{Inv}(\gamma, \mathtt{src}) \land \gamma^{\mathsf{s}}(\mathtt{i}){=}t \land \gamma^{\mathsf{r}}(\mathtt{i}){=}\bullet)}_I\right\}$
      cost = c[i]; $\left\{t * \boxed{\exists\gamma.\, \mathsf{g}(\gamma) * (\gamma_0 \dot{\cong} \gamma \land \mathsf{Inv}(\gamma, \mathtt{src}) \land \gamma^{\mathsf{s}}(\mathtt{i}){=}t \land \gamma^{\mathsf{r}}(\mathtt{i}){=}\bullet \land \mathtt{cost}{=}\gamma^{\mathsf{c}}(\mathtt{i}))}_I\right\}$
4.    for(j=0;j<size;j++){
   $\left\{t * \boxed{\exists\gamma.\, \mathsf{g}(\gamma) * (\gamma_0 \dot{\cong} \gamma \land \mathsf{Inv}(\gamma, \mathtt{src}) \land \gamma^{\mathsf{s}}(\mathtt{i}){=}t \land \gamma^{\mathsf{r}}(\mathtt{i}){=}1^{\mathtt{j}}.0^{\mathtt{size-j}} \land \mathtt{cost}{=}\gamma^{\mathsf{c}}(\mathtt{i}))}_I\right\}$
       newcost = cost + a[i][j]; b = 1;
   $\left\{t * \boxed{\begin{array}{l}\exists\gamma.\, \mathsf{g}(\gamma) * (\gamma_0 \dot{\cong} \gamma \land \mathsf{Inv}(\gamma, \mathtt{src}) \land \gamma^{\mathsf{s}}(\mathtt{i}){=}t \land \gamma^{\mathsf{r}}(\mathtt{i}){=}1^{\mathtt{j}}.0^{\mathtt{size-j}} \\ \land \mathtt{cost}{=}\gamma^{\mathsf{c}}(\mathtt{i}) \land \mathtt{newcost}{=}\mathtt{cost}{+}\gamma^{\mathrm{E}}[\mathtt{i}][\mathtt{j}] \land \mathtt{b}{=}1)\end{array}}_I\right\}$
5.     do{ oldcost=c[j];  //perform $A_t^2$
   $\left\{t * \boxed{\begin{array}{l}\exists\gamma, c.\mathsf{g}(\gamma) * (\gamma_0 \dot{\cong} \gamma \land \mathsf{Inv}(\gamma, \mathtt{src}) \land \gamma^{\mathsf{s}}(\mathtt{i}){=}t \land \gamma^{\mathsf{r}}(\mathtt{i}){=}1^{\mathtt{j}}.c.0^{\mathtt{size-j-1}} \\ \land \mathtt{cost}{=}\gamma^{\mathsf{c}}(\mathtt{i}) \land \mathtt{newcost}{=}\mathtt{cost}{+}\gamma^{\mathrm{E}}[\mathtt{i}][\mathtt{j}] \land \mathtt{b}{=}1 \land \mathtt{oldcost}{=}c)\end{array}}_I\right\}$
6.      if(newcost<oldcost){
   $\left\{t * \boxed{\begin{array}{l}\exists\gamma, c.\, \mathsf{g}(\gamma) * (\gamma_0 \dot{\cong} \gamma \land \mathsf{Inv}(\gamma, \mathtt{src}) \land \gamma^{\mathsf{s}}(\mathtt{i}){=}t \land \gamma^{\mathsf{r}}(\mathtt{i}){=}1^{\mathtt{j}}.c.0^{\mathtt{size-j-1}} \\ \land \mathtt{cost}{=}\gamma^{\mathsf{c}}(\mathtt{i}) \land \mathtt{newcost}{=}\mathtt{cost}{+}\gamma^{\mathrm{E}}[\mathtt{i}][\mathtt{j}] \land \mathtt{oldcost}{=}c \land \mathtt{newcost}{<}\mathtt{oldcost})\end{array}}_I\right\}$
7.       b=<CAS(work[j],1,0)>;   //perform $A_t^3$ if possible
8.       if(b){
   $\left\{t * \boxed{\begin{array}{l}\exists\gamma, c.\, \mathsf{g}(\gamma) * (\gamma_0 \dot{\cong} \gamma \land \mathsf{Inv}(\gamma, \mathtt{src}) \land \gamma^{\mathsf{s}}(\mathtt{i}){=}t \land \gamma^{\mathsf{r}}(\mathtt{i}){=}1^{\mathtt{j}}.c.0^{\mathtt{size-j-1}} \land \mathtt{cost}{=}\gamma^{\mathsf{c}}(\mathtt{i}) \\ \land \mathtt{newcost}{=}\mathtt{cost}{+}\gamma^{\mathrm{E}}[\mathtt{i}][\mathtt{j}] \land \mathtt{oldcost}{=}c \land \mathtt{newcost}{<}\mathtt{oldcost} \land \gamma^{\mathsf{s}}(\mathtt{j}){=}t \land \gamma^{\mathsf{r}}(\mathtt{j}){=}\bigcirc)\end{array}}_I\right\}$
        b=<CAS(c[j],oldcost,newcost)>;   //perform $A_t^4$ if possible
   $\left\{t * \boxed{\begin{array}{l}\exists\gamma, c.\, \mathsf{g}(\gamma) * (\gamma_0 \dot{\cong} \gamma \land \mathsf{Inv}(\gamma, \mathtt{src}) \land \gamma^{\mathsf{s}}(\mathtt{i}){=}t \land \mathtt{cost}{=}\gamma^{\mathsf{c}}(\mathtt{i}) \land \mathtt{newcost}{<}\mathtt{oldcost} \\ \land ((\mathtt{b}{=}1 \land \gamma^{\mathsf{r}}(\mathtt{i}){=}1^{\mathtt{j+1}}.0^{\mathtt{size-j-1}}) \lor (\mathtt{b}{=}0 \land \gamma^{\mathsf{r}}(\mathtt{i}){=}1^{\mathtt{j}}.c.0^{\mathtt{size-j-1}})) \land \gamma^{\mathsf{s}}(\mathtt{j}){=}t \land \gamma^{\mathsf{r}}(\mathtt{j}){=}\bigcirc)\end{array}}_I\right\}$
        <work[j] = 1>; }  //perform $A_t^5$ or $A_t^6$ depending on the value of b
   $\left\{t * \boxed{\begin{array}{l}\exists\gamma.\, \mathsf{g}(\gamma) * (\gamma_0 \dot{\cong} \gamma \land \mathsf{Inv}(\gamma, \mathtt{src}) \land \gamma^{\mathsf{s}}(\mathtt{i}){=}t \land \mathtt{cost}{=}\gamma^{\mathsf{c}}(\mathtt{i}) \land \mathtt{newcost}{<}\mathtt{oldcost} \\ \land ((\mathtt{b}{=}1 \land \gamma^{\mathsf{r}}(\mathtt{i}){=}1^{\mathtt{j+1}}.0^{\mathtt{size-j-1}}) \lor (\mathtt{b}{=}0 \land \gamma^{\mathsf{r}}(\mathtt{i}){=}1^{\mathtt{j}}.0^{\mathtt{size-j}})))\end{array}}_I\right\}$
```

Figure 7: A proof sketch of the `dijkstra` algorithm (continued in Fig. 8)

$$\mathsf{resp}(\gamma) \overset{\text{def}}{=} \mathop{\bigstar}_{i\in\gamma} \Big(\mathop{\bigstar}_{j\in\gamma} r[i][j] \Rightarrow \gamma^{\mathsf{r}}[i][j]\Big) \qquad\qquad \mathsf{state}(\gamma) \overset{\text{def}}{=} \mathop{\bigstar}_{i\in\gamma} \big(s[i] \Rightarrow \gamma^{\mathsf{s}}(i)\big)$$

We specify the spatial precondition of `dijkstra`, $\mathsf{Pre}(t,\gamma_0)$, as a CoLoSL assertion defined below where t identifies the running thread, and γ_0 denotes the original graph (at the beginning of `parallel_dijkstra`, before spawning new threads). We instantiate the CoLoSL capabilities to be the same as our tokens. The precondition Pre states that the current thread t holds the t capability, that the original graph γ_0 may have evolved to another congruent graph γ (captured by the existential quantifier) satisfying the invariant Inv, and that the shared state contains the spatial resources of the graph $\mathsf{g}(\gamma)$. As before, the spatial actions on the shared state are declared in I by lifting mathematical actions to spatial ones indexed by the corresponding capability. Finally, the spatial postcondition Post

```
9.        else {
    $\left\{t * \boxed{\begin{array}{l}\exists\gamma,c.\, \mathsf{g}(\gamma) * (\gamma_0 \dot{\cong} \gamma \wedge \mathsf{Inv}(\gamma, \mathtt{src}) \wedge \gamma^{\mathsf{s}}(\mathtt{i}){=}t \wedge \gamma^{\mathsf{r}}(\mathtt{i}){=}1^{\mathtt{j}}.c.0^{\mathtt{size-j-1}} \wedge \mathtt{cost}{=}\gamma^{\mathsf{c}}(\mathtt{i}) \\ \wedge\, \mathtt{newcost}{=}\mathtt{cost}{+}\gamma^{\mathrm{E}}[\mathtt{i}][\mathtt{j}] \wedge \mathtt{oldcost}{=}c \wedge \mathtt{newcost}{<}\mathtt{oldcost})\end{array}}_I\right\}$
           b=<CAS(done[j],1,0)>;    $//\text{perform } A_t^3 \text{ if possible}$
10.        if(b){
    $\left\{t * \boxed{\begin{array}{l}\exists\gamma,c.\, \mathsf{g}(\gamma) * (\gamma_0 \dot{\cong} \gamma \wedge \mathsf{Inv}(\gamma, \mathtt{src}) \wedge \gamma^{\mathsf{s}}(\mathtt{i}){=}t \wedge \gamma^{\mathsf{r}}(\mathtt{i}){=}1^{\mathtt{j}}.c.0^{\mathtt{size-j-1}} \wedge \mathtt{cost}{=}\gamma^{\mathsf{c}}(\mathtt{i}) \\ \wedge\, \mathtt{newcost}{=}\mathtt{cost}{+}\gamma^{\mathrm{E}}[\mathtt{i}][\mathtt{j}] \wedge \mathtt{oldcost}{=}c \wedge \mathtt{newcost}{<}\mathtt{oldcost} \wedge \gamma^{\mathsf{s}}(\mathtt{j}){=}t \wedge \gamma^{\mathsf{r}}(\mathtt{j}){=}\bullet)\end{array}}_I\right\}$
             b=<CAS(c[j],oldcost,newcost)>;    $//\text{perform } A_t^4 \text{ if possible}$
    $\left\{t * \boxed{\begin{array}{l}\exists\gamma,c.\, \mathsf{g}(\gamma) * (\gamma_0 \dot{\cong} \gamma \wedge \mathsf{Inv}(\gamma, \mathtt{src}) \wedge \gamma^{\mathsf{s}}(\mathtt{i}){=}t \wedge \mathtt{cost}{=}\gamma^{\mathsf{c}}(\mathtt{i}) \wedge \mathtt{newcost}{<}\mathtt{oldcost} \\ \wedge\, ((\mathtt{b}{=}1 \wedge \gamma^{\mathsf{r}}(\mathtt{i}){=}1^{\mathtt{j+1}}.0^{\mathtt{size-j-1}}) \vee (\mathtt{b}{=}0 \wedge \gamma^{\mathsf{r}}(\mathtt{i}){=}1^{\mathtt{j}}.c.0^{\mathtt{size-j-1}})) \\ \wedge\, \mathtt{newcost}{=}\mathtt{cost}{+}\gamma^{\mathrm{E}}[\mathtt{i}][\mathtt{j}] \wedge \gamma^{\mathsf{s}}(\mathtt{j}){=}t \wedge \gamma^{\mathsf{r}}(\mathtt{j}){=}\bullet)\end{array}}_I\right\}$
11.          if(b){<work[j]=1>}else{<done[j]=1>} $//A_t^5 \text{ or } A_t^7 \text{ based on}$ b
    $\left\{t * \boxed{\begin{array}{l}\exists\gamma.\, \mathsf{g}(\gamma) * (\gamma_0 \dot{\cong} \gamma \wedge \mathsf{Inv}(\gamma, \mathtt{src}) \wedge \gamma^{\mathsf{s}}(\mathtt{i}){=}t \wedge \mathtt{cost}{=}\gamma^{\mathsf{c}}(\mathtt{i}) \wedge \mathtt{newcost}{<}\mathtt{oldcost} \\ \wedge\, ((\mathtt{b}{=}1 \wedge \gamma^{\mathsf{r}}(\mathtt{i}){=}1^{\mathtt{j+1}}.0^{\mathtt{size-j-1}}) \vee (\mathtt{b}{=}0 \wedge \gamma^{\mathsf{r}}(\mathtt{i}){=}1^{\mathtt{j}}.0^{\mathtt{size-j}})))\end{array}}_I\right\}$
12.      }}}
    $\left\{t * \boxed{\begin{array}{l}\exists\gamma.\mathsf{g}(\gamma) * (\gamma_0 \dot{\cong} \gamma \wedge \mathsf{Inv}(\gamma, \mathtt{src}) \wedge \gamma^{\mathsf{s}}(\mathtt{i}){=}t \\ ((\mathtt{newcost}{<}\mathtt{oldcost} \wedge \mathtt{b}{=}1 \wedge \gamma^{\mathsf{r}}(\mathtt{i}){=}1^{\mathtt{j+1}}.0^{\mathtt{size-j-1}}) \\ \vee (\mathtt{newcost}{<}\mathtt{oldcost} \wedge \mathtt{b}{=}0 \wedge \gamma^{\mathsf{r}}(\mathtt{i}){=}1^{\mathtt{j}}.0^{\mathtt{size-j}}) \\ \vee (\mathtt{newcost}{\geqslant}\mathtt{oldcost} \wedge \mathtt{b}{=}1 \wedge \gamma^{\mathsf{r}}(\mathtt{i}){=}1^{\mathtt{j}}.-.0^{\mathtt{size-j-1}}))\end{array}}_I\right\}$ $//\text{perform } A_t^8 \text{ on 3rd disjunct}$
    $\left\{t * \boxed{\begin{array}{l}\exists\gamma.\, \mathsf{g}(\gamma) * (\gamma_0 \dot{\cong} \gamma \wedge \mathsf{Inv}(\gamma, \mathtt{src}) \wedge \gamma^{\mathsf{s}}(\mathtt{i}){=}t \\ ((\mathtt{b}{=}1 \wedge \gamma^{\mathsf{r}}(\mathtt{i}){=}1^{\mathtt{j+1}}.0^{\mathtt{size-j-1}}) \vee (\mathtt{b}{=}0 \wedge \gamma^{\mathsf{r}}(\mathtt{i}){=}1^{\mathtt{j}}.0^{\mathtt{size-j}}))\end{array}}_I\right\}$
13.     } while(!b) $\left\{t * \boxed{\exists\gamma.\, \mathsf{g}(\gamma) * (\gamma_0 \dot{\cong} \gamma \wedge \mathsf{Inv}(\gamma, \mathtt{src}) \wedge \gamma^{\mathsf{s}}(\mathtt{i}){=}t \wedge \gamma^{\mathsf{r}}(\mathtt{i}){=}1^{\mathtt{j+1}}.0^{\mathtt{size-j-1}})}_I\right\}$
14.    } $\left\{t * \boxed{\exists\gamma.\, \mathsf{g}(\gamma) * (\gamma_0 \dot{\cong} \gamma \wedge \mathsf{Inv}(\gamma, \mathtt{src}) \wedge \gamma^{\mathsf{s}}(\mathtt{i}){=}t \wedge \gamma^{\mathsf{r}}(\mathtt{i}){=}\bullet)}_I\right\}$
       <done[i]=1>;    $//\text{perform } A_t^9$  $\left\{t * \boxed{\exists\gamma.\, \mathsf{g}(\gamma) * (\gamma_0 \dot{\cong} \gamma \wedge \mathsf{Inv}(\gamma, \mathtt{src}) \wedge \gamma^{\mathsf{s}}(\mathtt{i}){=}1)}_I\right\}$
15.   } i = (i+1) mod size;
16. } } $\left\{t * \boxed{\exists\gamma.\, \mathsf{g}(\gamma) * (\gamma_0 \dot{\cong} \gamma \wedge \mathsf{Inv}(\gamma, \mathtt{src}) \wedge \forall x.\, \gamma^{\mathsf{s}}(x){=}1)}_I\right\}$
$\{\mathsf{Post}(t,\gamma_0)\}$
```

Figure 8: A proof sketch of the `dijkstra` algorithm (continued from Fig. 7)

is analogous to Pre and further states that all nodes in γ are processed (in `done`).

$$\mathsf{Pre}(t,\gamma_0) \stackrel{\text{def}}{=} t * \boxed{\exists\gamma.\, \mathsf{g}(\gamma) * (\gamma_0 \doteq \gamma \wedge \mathsf{Inv}(\gamma, \mathtt{src}))}_I \qquad I \stackrel{\text{def}}{=} \{t\colon \mathsf{g}(\gamma) \wedge \gamma\, A_t\, \gamma' \rightsquigarrow \mathsf{g}(\gamma')$$

$$\mathsf{Post}(t,\gamma_0) \stackrel{\text{def}}{=} t * \boxed{\exists\gamma.\, \mathsf{g}(\gamma) * (\gamma_0 \doteq \gamma \wedge \mathsf{Inv}(\gamma, \mathtt{src}) \wedge \forall x \in \gamma.\, \gamma^{\mathsf{s}}(x) \doteq 1)}_I$$

Verifying `parallel_dijkstra` A proof sketch of `dijkstra` is given in Figs. 7-8. As before, in all proof points the spatial part ($\mathsf{g}(\gamma)$) remains unchanged, and the changes to the graph are reflected in the changes to the pure mathematical assertions. Observe that when all threads return, the pure part of the postcondition ($\mathsf{Inv}(\gamma, \mathtt{src}) \wedge \forall x \in \gamma.\, \gamma^{\mathsf{s}}(x) \doteq 1$) entails that all costs in `cost` are minimal as per the first (and the only applicable) disjunct in $\mathsf{Inv}(\gamma, \mathtt{src})$. As such, the proof of `parallel_dijkstra` is immediate from the parallel rule (Par).

Concluding remarks We have verified two sophisticated concurrent graph algorithms, `copy_dag` and `parallel_dijkstra`, neither of which has been verified previously. We used several proof patterns, such as doing the tricky reasoning on mathematical abstractions and using tokens to track the progress of cooperating threads. We used an "iCAP-like" abstract proof style despite using CoLoSL which does not support this proof style natively. In [10, 11] we verify two further graph algorithms using our proof pattern: graph marking, which is the simplest nontrivial concurrent algorithm and which accordingly enjoys the cleanest proof; and spanning tree, which has been done previously but with different invariants.

Acknowledgements This research was supported by EPSRC programme grants EP/H008373/1 and EP/K008528/1, Yale-NUS College and R-607-265-045-121.

References

1. R. Bornat, C. Calcagno, and P. O'Hearn. Local reasoning, separation and aliasing. In *SPACE*, volume 4, 2004.
2. E. Dijkstra, L. Lamport, A. Martin, C. Scholten, and E. Steffens. On-the-fly garbage collection: an exercise in cooperation. In Friedrich L. Bauer and Klaus Samelson, editors, *Language Hierarchies and Interfaces*, volume 46 of *Lecture Notes in Computer Science*, pages 43–56. Springer, 1975.
3. T. Dinsdale-Young, L. Birkedal, P. Gardner, M. Parkinson, and H. Yang. Views: compositional reasoning for concurrent programs. In *POPL*, pages 287–300, 2013.
4. T. Dinsdale-Young, M. Dodds, P. Gardner, M. Parkinson, and V. Vafeiadis. Concurrent abstract predicates. In *ECOOP*, pages 504–528, 2010.
5. R. Dockins, A. Hobor, and A. Appel. A fresh look at separation algebras and share accounting. In *APLAS*, 2009.
6. X. Feng. Local rely-guarantee reasoning. In *POPL*, pages 315–327, 2009.
7. A. Grama, G. Anshul, G. Karypis, and V. Kumar. *Introduction to Parallel Computing (Second Ed.).* Addison Wesley, 2003.
8. A. Hobor and J. Villard. The ramifications of sharing in data structures. In Roberto Giacobazzi and Radhia Cousot, editors, *POPL*, pages 523–536. ACM, 2013.
9. A. Nanevski, R. Ley-Wild, I. Sergey, and G. Delbianco. Communicating state transition systems for fine-grained concurrent resources. In *ESOP*, pages 290–310, 2014.
10. A. Raad. *(To appear)*. PhD thesis, Imperial College London, 2016.
11. A. Raad, A. Hobor, J. Villard, and P. Gardner. Verifying concurrent graph algorithms (extended). 2016.
12. A. Raad, J. Villard, and P. Gardner. CoLoSL: Concurrent Local Subjective Logic. In *ESOP*, pages 710–735, 2015.
13. J. Reynolds. A short course on separation logic. `http://www.cs.cmu.edu/afs/cs.cmu.edu/project/fox-19/member/jcr/wwwaac2003/notes7.ps`, 2003.
14. I. Sergey, A. Nanevski, and A. Banerjee. Mechanized verification of fine-grained concurrent programs. In *PLDI*, 2015.
15. K. Svendsen and L. Birkedal. Impredicative concurrent abstract predicates. In *ESOP*, 2014.
16. A. Turon, D. Dreyer, and L. Birkedal. Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency. In *ICFP*, pages 377–390, 2013.
17. H. Yang. *Local Reasoning for Stateful Programs.* PhD thesis, University of Illinois, 2001.