Verifying Concurrent Graph Algorithms

Azalea Raad[†], Aquinas Hobor[‡], Philippa Gardner[†], Jules Villard[†]
[†]Imperial College London [‡]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 and spanning tree. We develop a method of reasoning 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 abstract reasoning in the style of iCAP without needing to incorporate any associated support in the underlying semantic model of the logic.

1 Introduction

Verifying fine-grained concurrent algorithms is nontrivial. There has been much progress recently in verifying such algorithms modularly using variants of concurrent separation logic [10, 12, 16, 15, 7, 5]. One area of particular difficulty has been verifying such algorithms that manipulate graphs. This is only to be expected: even in a semiformal "algorithmic" sense, the correctness arguments of concurrent graph algorithms can be dauntingly subtle [3].

To verify such algorithms, we must not only understand these algorithmic arguments but also must 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. 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 rectify this lack by verifying the functional correctness of four nontrivial concurrent fine-grained graph-manipulating algorithms. We study a structure-preserving copy, a speculatively-parallel version of Dijkstra's shortest-path algorithm, a marking algorithm, and an algorithm to prune a graph into a spanning tree. We have found some 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 specify the behavior of each thread using a notion of *tokens* to track each thread's portion of the computation. Informally, if thread t's token is on vertex v, then t is responsible for some work on/around v. Our tokens are sufficiently general to be able to handle sophisticated parallelism, e.g. when the algorithm wishes to create or destroy threads when the underlying graph splits or merges.

We then reason about the memory safety of the algorithm by lifting 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. $\mathsf{graph}(\gamma) \stackrel{\text{def}}{=} \ldots$. We define our spatial predicates in such a way that simplifies many of the proof obligations (e.g. when parallel computations join).

Our pattern of doing the bulk of the tricky reasoning on abstract structures is similar to the style used in program logics such as iCAP [15] and CaReSL [16]. Just as with these logics, carrying out the reasoning at an abstract level makes the proofs simpler and more readable, and lessens the burden of side conditions required by concurrent program logics such as establishing stability. This abstract style of reasoning is "baked in" to the underlying semantic model of theses logics. Interestingly, we show that this baking is unnecessary by using a logic (CoLoSL [12]) without such built-in support, hinting that the underlying semantic models for logics such as iCAP and CaReSL may be more complicated than necessary. In fact we do not use any of the unique features of ColoSL and have verified the same algorithms in another program logic [11], giving us confidence that our verifications should port to other 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 in [9]. We make critical use of some of the graph-related verification infrastructure in [9].

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.

For concurrent graph algorithms, both Raad et al. [12] and Sergey et al. [14] have verified a concurrent spanning tree algorithm, one of our examples. Although we use the program logic CoLoSL developed in [12], the verification of spanning tree given there was only shape-based, whereas our proof here is for full functional correctness. The proof in [14] was for full functional correctness 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.

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. In appendix A we verify two further graph algorithms, graph marking (§A.1) and spanning tree (§A.2).

2 Background

2.1 CoLoSL: Concurrent Local Subjective Logic

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

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

This assertion 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 the 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 $x{+}1$ 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: $\boxed{P}_{I \cup I'} \wedge$ [side-condition-omitted] $\stackrel{\text{SHIFT}}{\Longrightarrow} \boxed{P}_{I}$. Hence, subjective views may also arbitrarily overlap in their interference relations. For space reason we have omitted this rule from Fig. 1 as we do not use it here. 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* [9], $\circledast$, 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 (see §A.1).

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, $\mathbb{H} \stackrel{\text{def}}{=} (\mathcal{H}, \uplus, \varnothing)$, with the composition operator as the disjoint function union, and the function with empty

[1] CoLoSL stipulates that PCMs satisfy the cross-split property [9], which ours do.

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

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

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

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 [4], with $\Rrightarrow$ denoting the *semantic consequence* relation. 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 through an application of Con, without the need for 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 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 an algorithm with only two threads this might be as simple as the set $\{\mathsf{red}, \mathsf{blue}\}$.

Third, we define the *mathematical actions* of the algorithm 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. 3 makes a deep structure-preserving copy of the dag (directed acyclic graph) rooted at `x` concurrently. To do this, each

node `x` in the original 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. Conversely, if `x` has not been copied yet, the thread allocates a new node `y` to serve as its copy 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 [9]. However, the non-deterministic behaviour of parallel computation makes even *specifying* the behaviour of `copy_dag` challenging. Observe that each node x of the original 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' then visits x, it simply returns even though x and its children may not have been fully copied yet. The challenge then is: how do we specify the postcondition of thread π' since we cannot promise that the subdag at x is fully copied when it returns? Intuitively, the reason that thread π' can safely return in this case is 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. $(x,0)$ in 2a; ii) a grey ellipse when in stage 2 e.g. $x, x' \mid \pi$ in 2b where thread π has copied x to x'; and iii) a black circle when in stage 3, e.g. x, x' in 2e. 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 an other 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 sub-thread (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. 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 [6] as described below.

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

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

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 $\circ\widehat{\ }\bullet$ and $\bullet\widehat{\ }\circ$, respectively. We model the ancestor-descendant relation between threads by the $\sqsubset$ ordering defined below where $+$ denotes the transitive closure of the relation.

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

We write $\pi \sqsubseteq \pi'$ for $\pi = \pi' \vee \pi \sqsubset \pi'$. Finally, we 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. $\overline{\pi} \stackrel{\text{def}}{=} \{\pi' \mid \pi' \sqsubseteq \pi\}$.

As discussed in §2.2, we carry out most of our reasoning abstractly by appealing to *mathematical objects* that provide an *abstract* representation of the shared data structure. 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. $\delta_1 \cong \delta_2 \stackrel{\text{def}}{=} \delta_1^{\mathrm{V}}{=}\delta_2^{\mathrm{V}} \wedge \delta_1^{\mathrm{E}}{=}\delta_2^{\mathrm{E}}$. 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 original 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 above: 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 set values 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 *unprocessed path* relation, $x \overset{\delta}{\rightsquigarrow}_0 y$, as follows and write $\overset{\delta}{\rightsquigarrow}_0^\star$ to describe the reflexive transitive closure of $\overset{\delta}{\rightsquigarrow}_0$.

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

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 y and takes x to stage 2 ($c{=}y$, $s{=}\pi$). Once π finishes executing line 10, it takes x to stage 3 ($c{=}y$, $s{=}0$). If another thread π' then visits x when it is in stage 2 or 3, it

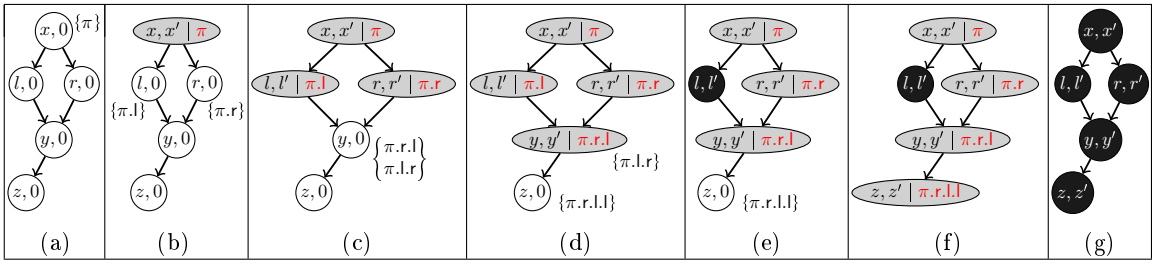


Figure 2: An example trace of `copy_dag`.

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 first set of actions, A^1_π, describes taking a node x from stage 1 to 2 by thread π (i.e. the atomic `CAS` instruction of lines 5-7 when successful). 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.

$$A^1_\pi \stackrel{\text{def}}{=} \left\{ \left(\begin{array}{l} (\delta_1,\delta_2),\\ (\delta_1',\delta_2') \end{array} \middle| \begin{array}{l} \delta_1(x){=}\big((0{,}0{,}P{\uplus}\{\pi\}),l,r\big) \wedge \delta_1{}^{\mathsf{L}}(l){=}(c_l,s_l,P_l) \wedge \delta_1{}^{\mathsf{L}}(r){=}(c_r,s_r,P_r)\\ {\wedge}\delta_1' = (\delta_1^{\mathsf{V}},\delta_1^{\mathsf{E}},L_1') \wedge \delta_2' = (V_2',E_2',L_2')\\ {\wedge}L_1'{=}\delta_1^{\mathsf{L}}[r\mapsto c_r,s_r,P_r{\uplus}\{\pi.\mathsf{r}\}][l\mapsto c_l,s_l,P_l\uplus\{\pi.\mathsf{l}\}][x\mapsto(y,\pi,P)]\\ {\wedge}V_2'{=}\delta_2^{\mathsf{V}}{\uplus}\{y\} \wedge E_2'{=}\delta_2^{\mathsf{E}}{\uplus}[y\mapsto(0{,}0)] \wedge L_2'{=}\delta_2^{\mathsf{L}}{\uplus}[y\mapsto(0{,}\pi{,}\varnothing)] \end{array} \right) \right\}$$

The next sets of actions correspond to 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.

$$A^2_\pi \stackrel{\text{def}}{=} \left\{ \left(\begin{array}{l} (\delta_1,\delta_2),\\ (\delta_1,\delta_2') \end{array} \middle| \begin{array}{l} \delta_1(x){=}\big((y,\pi,P),l,-\big) \wedge (l{=}0\wedge c_l{=}0 \vee \delta_1^{\mathsf{c}}(l){=}c_l\wedge c_l{\neq}0)\\ {\wedge}\delta_2(y){=}\big((0,\pi,\varnothing),-,r\big) \wedge \delta_2'{=}(\delta_2^{\mathsf{V}},E_2',\delta_2^{\mathsf{L}}) \wedge E_2'{=}[\delta_2^{\mathsf{E}}y\mapsto(c_l,r)] \end{array} \right) \right\}$$

$$A^3_\pi \stackrel{\text{def}}{=} \left\{ \left(\begin{array}{l} (\delta_1,\delta_2),\\ (\delta_1,\delta_2') \end{array} \middle| \begin{array}{l} \delta_1(x){=}\big((y,\pi,P),-,r\big) \wedge (r{=}0\wedge c_r{=}0 \vee \delta_1^{\mathsf{c}}(r){=}c_r\wedge c_r{\neq}0)\\ {\wedge}\delta_2(y){=}\big((0,\pi,\varnothing),l,-\big) \wedge \delta_2'{=}(\delta_2^{\mathsf{V}},E_2',\delta_2^{\mathsf{L}}) \wedge E_2'{=}\delta_2^{\mathsf{E}}[y\mapsto(l,c_r)] \end{array} \right) \right\}$$

$$A^4_\pi \stackrel{\text{def}}{=} \left\{ \left(\begin{array}{l} (\delta_1,\delta_2),\\ (\delta_1',\delta_2') \end{array} \middle| \begin{array}{l} \delta_1(x){=}\big((y,\pi,P),l,r\big) \wedge \delta_2(y){=}\big((0,\pi,\varnothing),c_l,c_r\big)\\ {\wedge}(l{=}0\wedge c_l{=}0 \vee \delta_1^{\mathsf{c}}(l){=}c_l\wedge c_l{\neq}0) \wedge (r{=}0\wedge c_r{=}0 \vee \delta_1^{\mathsf{c}}(r){=}c_r\wedge c_r{\neq}0)\\ {\wedge}\delta_1'{=}(\delta_1^{\mathsf{V}},\delta_1^{\mathsf{E}},\delta_1^{\mathsf{L}}[x{\mapsto}(y{,}0{,}P)]) \wedge \delta_2'{=}(\delta_2^{\mathsf{V}},\delta_2^{\mathsf{E}},\delta_2^{\mathsf{L}}[y{\mapsto}(0{,}0{,}\varnothing)]) \end{array} \right) \right\}$$

The last set of actions 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.

$$A^5_\pi \stackrel{\text{def}}{=} \left\{ \big((\delta_1,\delta_2),(\delta_1',\delta_2)\big) \,\middle|\, \delta_1^{\mathsf{L}}(x){=}(y,s,P{\uplus}\{\pi\}) \wedge y{\neq}0 \wedge \delta_1'{=}\big(\delta_1^{\mathsf{V}},\delta_1^{\mathsf{E}},\delta_1^{\mathsf{L}}[x\mapsto(y,s,P)]\big) \right\}$$

We write A_π to denote the actions of thread π: $A_\pi \stackrel{\text{def}}{=} A^1_\pi \cup A^2_\pi \cup A^3_\pi \cup A^4_\pi \cup A^5_\pi$. We now have all ingredients necessary to specify the behaviour of `copy_dag` mathematically.

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

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

$$\begin{aligned}\mathsf{ic}(x,x',\delta,\delta') \stackrel{\text{def}}{=}\ & (x'=0 \wedge \exists y.\, \delta^{\mathsf{p}}(y) \neq \varnothing \wedge y \overset{\delta}{\rightsquigarrow}_0^* x) \vee (x' \neq 0 \wedge x' \in \delta' \wedge \exists \pi.\, \delta^{\mathsf{s}}(x)=\pi) \\ & \vee \big(x' \neq 0 \wedge x' \in \delta' \wedge \exists l, r, l', r'.\ \delta(x)=((x',0,-),l,r) \\ & \qquad \wedge \delta'^{\mathsf{l}}(x')=l' \wedge \mathsf{ic}(l,l',\delta,\delta') \wedge \delta'^{\mathsf{r}}(x')=r' \wedge \mathsf{ic}(r,r',\delta,\delta')\big)\end{aligned}$$

Informally, the invariant asserts that each node x' of the copy dag δ' corresponds to a unique node x of the original dag δ (first conjunct). The second conjunct states that each node x of the original dag is in one of the three stages described above, via the ic predicate: i) x is not copied yet (stage 1), in which case there is a path from a node y with a non-empty promise set to x, ensuring that the node will eventually be visited (first disjunct); ii) x is currently being processed (stage 2) by thread π (second disjunct); 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 original 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 subthreads subsequently spawned by π. As such, 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). Finally, 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'). None of the descendants of π (including π itself) can be found in the promise sets. Similarly, π and all his descendants must have finished processing their charges and thus cannot correspond to the state field of any node.

$$\begin{aligned}\mathsf{P}^{\pi}(x,\delta) \stackrel{\text{def}}{=}\ & (x=0 \vee \pi \in \delta^{\mathsf{p}}(x)) \wedge \forall \pi'.\, \forall y \in \delta. \\ & \qquad (\pi' \in \delta^{\mathsf{p}}(y) \Rightarrow \pi' \sharp \pi) \wedge (x \neq y \Rightarrow \pi \notin \delta^{\mathsf{p}}(y)) \wedge (\delta^{\mathsf{s}}(y)=\pi' \Rightarrow \pi' \sharp \pi) \\ \mathsf{Q}^{\pi}(x,y,\delta,\delta') \stackrel{\text{def}}{=}\ & (x=0 \vee (\delta^{\mathsf{c}}(x)=y \wedge y \in \delta')) \wedge \forall \pi'.\, \forall z \in \delta. \\ & \qquad \pi' \in \delta^{\mathsf{p}}(z) \vee \delta^{\mathsf{s}}(z)=\pi' \Rightarrow \pi' \sharp \pi\end{aligned}$$

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. $\forall \pi.\, \pi \sqsubseteq \bullet$), $\mathsf{Q}^{\bullet}(\mathsf{x}, \mathsf{ret}, \delta, \delta')$ entails that no tokens can be found anywhere in δ, i.e. $\forall y.\, \delta^{\mathsf{p}}(y)=\varnothing \wedge \delta^{\mathsf{s}}(y)=0$. As such, $\mathsf{Q}^{\bullet}(\mathsf{x}, \mathsf{ret}, \delta, \delta')$ together with Inv entails that all nodes in δ have been correctly copied into δ', i.e. only the third disjunct of $\mathsf{ic}(\mathsf{x}, \mathsf{ret}, \delta, \delta')$ in Inv applies.

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*.*

Part (1) states that the invariant Inv is stable with respect to the actions of all threads (A_π for any tokens π). Parts (2) and (3) state that the pre- and post-conditions 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.

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. 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 left (l) and right (r) children, respectively; while 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 $\circledast$ (see §A.2). Here, we choose the implementation below for simplicity.

$$\mathsf{icdag}(\delta_1, \delta_2) \stackrel{\text{def}}{=} d \Rightarrow (\delta_1, \delta_2) * \mathsf{dag}(\delta_1) * \mathsf{dag}(\delta_2) \qquad \mathsf{dag}(\delta) \stackrel{\text{def}}{=} \bigstar_{x \in \delta} \mathsf{node}(x, \delta)$$

$$\mathsf{node}(x, \delta) \stackrel{\text{def}}{=} \exists l, r, c, s, P.\, \delta(x) = (c, s, P), l, r \wedge x \mapsto c, l, r * x \Rightarrow s, P$$

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 original 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 ($\overline{\pi}^\star$). 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 original dag 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 original 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 original 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.

$$\mathsf{Pre}(x, \pi, \delta) \stackrel{\text{def}}{=} \overline{\pi}^\star * \boxed{\exists \delta_1, \delta_2.\, \mathsf{icdag}(\delta_1, \delta_2) * (\delta \dot{\cong} \delta_1 \wedge \mathsf{Inv}(\delta_1, \delta_2) \wedge \mathsf{P}^\pi(x, \delta_1))}_I$$

$$\mathsf{Post}(x, y, \pi, \delta) \stackrel{\text{def}}{=} \overline{\pi}^\star * \boxed{\exists \delta_1, \delta_2.\, \mathsf{icdag}(\delta_1, \delta_2) * (\delta \dot{\cong} \delta_1 \wedge \mathsf{Inv}(\delta_1, \delta_2) \wedge \mathsf{Q}^\pi(x, y, \delta_1, \delta_2))}_I$$

$$\overline{\pi}^\star \stackrel{\text{def}}{=} \bigstar_{\pi \in \overline{\pi}} \pi \qquad I \stackrel{\text{def}}{=} \left\{ \pi : \mathsf{icdag}(\delta_1, \delta_2) \wedge (\delta_1, \delta_2) A_\pi (\delta_1', \delta_2') \rightsquigarrow \mathsf{icdag}(\delta_1', \delta_2') \right.$$

Verifying copy_dag We give a proof sketch of copy_dag in Fig. 3. 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 parts in the shared state: $\mathsf{icdag}(\delta_1, \delta_2)$. Indeed, the heap is changing constantly, due both to the actions of this thread and the environment. The spatial part states that 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 are handled by the existential quantifier.

On line 3 we check to see if x is 0. If so the program returns and the post-condition, $\mathsf{Post}(\mathsf{x}, 0, \delta, \pi)$, 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 we set b to false, perform action A_π^5 (i.e. remove π from the promise set of x) and thus arrive at the desired post-condition $\mathsf{Post}(\mathsf{x}, \delta_1^{\mathsf{c}}(\mathsf{x}), \pi, \delta)$. On the other hand, if x is not copied, we set b to true and perform the action A_π^1. 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 ($\boxed{P}_I \stackrel{\textsc{Copy}}{\Longrightarrow} \boxed{P}_I * \boxed{P}_I$), drop irrelevant pure assertions, and unwrap the definition of $\overline{\pi}^\star$. We then use the Par rule 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 ($\boxed{P}_I * \boxed{P}_I \stackrel{\textsc{Merge}}{\Longrightarrow} \boxed{P \curlywedge Q}_I \Rightarrow \boxed{P \wedge Q}_I$). Finally, on line 10 we perform

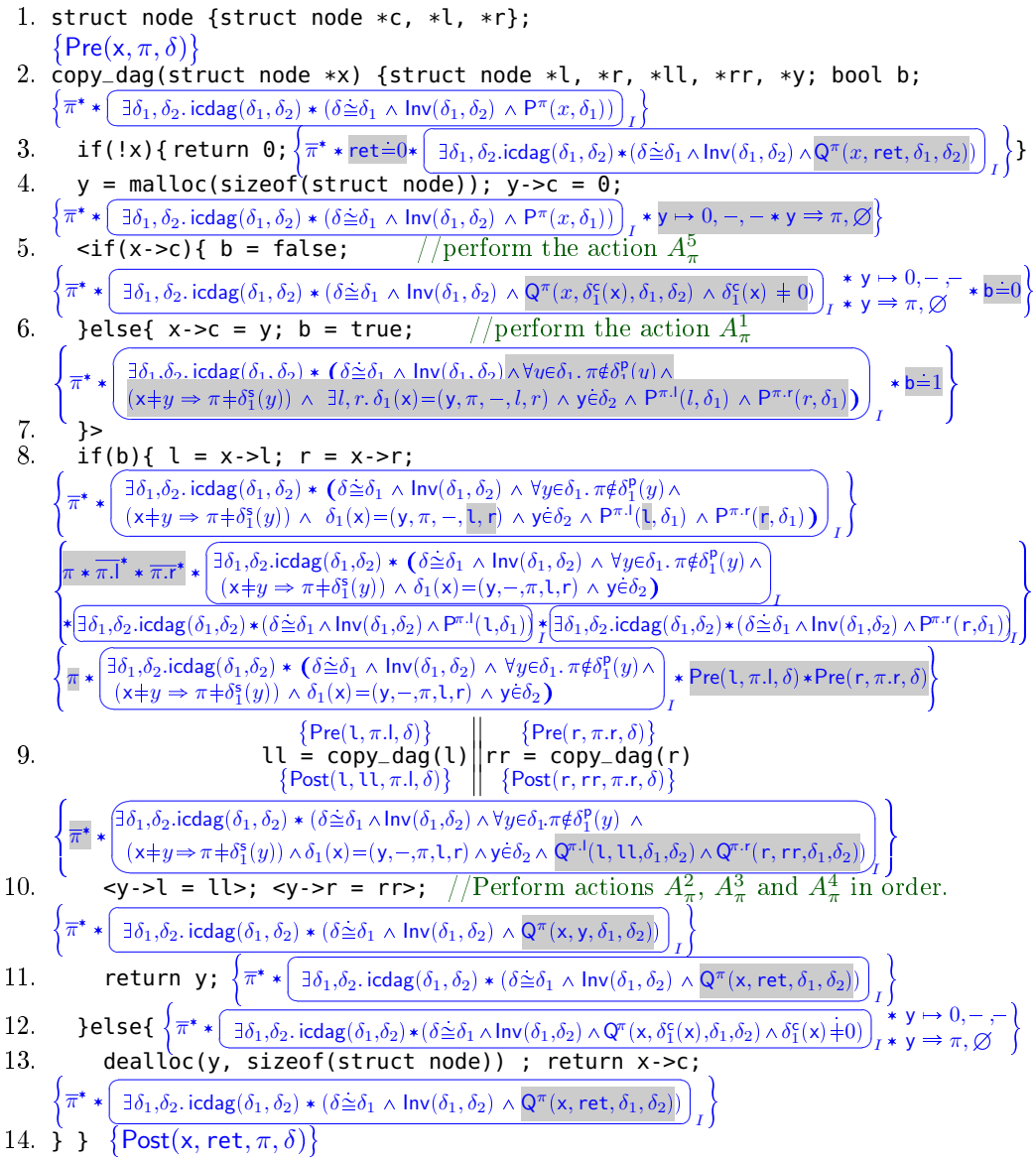


Figure 3: The code and a proof sketch of `copy_dag`.

actions A^2_π, A^3_π and A^4_π in order to update the edges of y, and arrive at the postcondition $\mathsf{Post}(\mathsf{x}, \mathsf{y}, \pi, \delta)$.

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 array for each node (bar the source itself) is initialised with the value of the adjacency matrix (i.e. `c[src]`=0; `c[i]`=`a[src][i]` for `i` $\neq$ `src`). Initially, all vertices are in `work` and the algorithm proceeds by iterating through `work` performing the following two steps in 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]`$\}$. The 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 algorithm, `parallel_dijkstra` in Fig. 4, 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. 6. 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 18-21). If on the other hand `j` is unprocessed (and is in `work`), we can safely decrease its cost (lines 13-16). 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* (speculative decomposition [8]) – each thread running `dijkstra` assumes that the costs associated with 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 shortest paths.

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, $E : V \rightarrow (V \rightarrow \mathcal{W})$ is the weighted adjacency function with weights $\mathcal{W} \stackrel{\text{def}}{=} \mathbb{N} \uplus \{\infty\}$, and $L : V \rightarrow D$ is the label function. 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 $t \in \Theta \stackrel{\text{def}}{=} \mathbb{N} \backslash \{0, 1\}$. We refer to the thread with token t simply as

```
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 }
```

Figure 4: A parallel non-greedy variant of Dijkstra's algorithm with `dijkstra` in Fig. 6.

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 $D \stackrel{\text{def}}{=} \mathcal{W} \times \big(\{0,1\} \uplus \Theta\big) \times (V \to \{\circ,\bullet\} \uplus \mathcal{W})$. The first component denotes the cost of 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 through all vertices examining whether their cost can be improved. To do this, at each iteration the thread records the current cost of node j under inspection in `oldcost` (line 11). If the cost may be improved (i.e. the conditional of line 12 succeeds), it then *attempts* to update the cost of j with the improved value (lines 15, 20). 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 O (resp. $\mathsf{\bullet}$) 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: $\gamma_1 \cong \gamma_2 \stackrel{\text{def}}{=} \gamma_1^{\mathrm{V}}{=}\gamma_2^{\mathrm{V}} \wedge \gamma_1^{\mathrm{E}}{=}\gamma_2^{\mathrm{E}}$. We define the weighted path relation ($\overset{\gamma}{\rightsquigarrow}_c$), and its reflexive transitive closure as:

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

Actions We define several families of actions in Fig. 5, each of which indexed by a token t. The A_t^1 set describes the action of line 5 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 set describes a *ghost* action at line 11 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 set describes the cases of lines 13 and 18 in the

$$
\begin{array}{l}
A_t^1 \stackrel{\text{def}}{=} \left\{ ((V,E,L),(V,E,L')) \mid L(i){=}(c,0,\circ) \wedge L'{=}L[i \mapsto (c,t,\circ)] \right\} \\
A_t^2 \stackrel{\text{def}}{=} \left\{ ((V,E,L),(V,E,L')) \left| \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. \right\} \\
A_t^3 \stackrel{\text{def}}{=} \left\{ ((V,E,L),(V,E,L')) \left| \begin{array}{l} L(i){=}(-,t,R) \wedge R[j]{=}c' \wedge L(j){=}(c,s,R') \wedge s{\in}\{0,1\} \\ \wedge\, L'{=}L[j{\mapsto}(c,t,R')] \end{array} \right. \right\} \\
A_t^4 \stackrel{\text{def}}{=} \left\{ ((V,E,L),(V,E,L')) \left| \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. \right\} \\
A_t^5 \stackrel{\text{def}}{=} \left\{ ((V,E,L),(V,E,L')) \mid L(i){=}(c,t,R) \wedge R[j]{=}\bullet \wedge L(j){=}(c',t,-) \wedge L'{=}L[j \mapsto (c',0,\circ)] \right\} \\
A_t^6 \stackrel{\text{def}}{=} \left\{ ((V,E,L),(V,E,L')) \left| \begin{array}{l} L(i){=}(c,t,R) \wedge R[j]{=}c'' \wedge L(j){=}(c',t,\circ) \wedge c'{\neq}c'' \\ \wedge\, R'{=}R[j{\mapsto}\circ] \wedge L'{=}L[i \mapsto (c,t,R')][j \mapsto (c',0,\circ)] \end{array} \right. \right\} \\
A_t^7 \stackrel{\text{def}}{=} \left\{ ((V,E,L),(V,E,L')) \left| \begin{array}{l} L(i){=}(c,t,R) \wedge R[j]{=}c'' \wedge L(j){=}(c',t,\bullet) \wedge c'{\neq}c'' \\ \wedge\, R'{=}R[j \mapsto \circ] \wedge L'{=}L[i \mapsto (c,t,R')][j \mapsto (c',1,\bullet)] \end{array} \right. \right\} \\
A_t^8 \stackrel{\text{def}}{=} \left\{ ((V,E,L),(V,E,L')) \left| \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. \right\} \\
A_t^9 \stackrel{\text{def}}{=} \left\{ ((V,E,L),(V,E,L')) \mid L(x){=}(c,t,\bullet) \wedge L'{=}[x \mapsto (c,1,\bullet)]L \right\}
\end{array}
$$

Figure 5: The mathematical actions of `dijkstra`.

algorithm: when processing i, we discovered that the cost of j may be improved. 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 set describes the `CAS` operations in lines 15 and 20 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 set denotes the reinsertion of j into `work` in lines 16 and 21 following *successful* `CAS` operations at lines 15 and 20. 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 16 and 21, following *failed* `CAS` operations at lines 15 and 20. 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 set describes a ghost action in line 12 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 set set captures the `CAS` operation in line 25: 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 thread t: $A_t \stackrel{\text{def}}{=} A_t^1 \cup A_t^2 \cup A_t^3 \cup A_t^4 \cup A_t^5 \cup A_t^6 \cup A_t^7 \cup A_t^8 \cup A_t^9$.

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

$$\mathsf{Inv}(\gamma, src) \stackrel{\text{def}}{=} \forall x \in \gamma.\, \mathsf{min}^{src}_{\gamma}(x, \gamma^{\mathsf{c}}(x)) \vee \Big(\exists y,z,c.\, \big(\gamma(y){=}0 \vee (\gamma(y){\neq}1 \wedge \gamma^{\mathsf{r}}[y][z]{=}0)\big) \wedge y \overset{\gamma}{\rightsquigarrow}_c z \wedge \mathsf{wit}^{src}_{\gamma}(\gamma^{\mathsf{c}}(y){+}c, z, x)\Big)$$

$$\mathsf{min}^{src}_{\gamma}(x, c) \stackrel{\text{def}}{=} min\{c' \mid s \overset{\gamma}{\rightsquigarrow}^{\star}_{c'} x\} = c$$

$$\mathsf{wit}^{src}_{\gamma}(c,z,x) \stackrel{\text{def}}{=} \mathsf{min}^{src}_{\gamma}(z, c) \wedge \gamma^{\mathsf{c}}(z) > c \wedge (z{=}x \vee (\exists c', w.\, z \overset{\gamma}{\rightsquigarrow}_{c'} w \wedge \mathsf{wit}^{src}_{\gamma}(c{+}c', w, x)))$$

$\mathsf{Inv}(\gamma, src)$ 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), where y is either unprocessed or being processed, and its cost is minimal. 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 and in `done` (i.e. $\forall x.\, \gamma^{\mathsf{s}}(x){=}1$), $\mathsf{Inv}(\gamma, src)$ 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:*

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

Proof. Follows from the definitions of A_t and Inv.

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

$$\mathsf{g}(\gamma) \stackrel{\text{def}}{=} \mathsf{work}(\gamma) * \mathsf{done}(\gamma) * \mathsf{adj}(\gamma) * \mathsf{cost}(\gamma) * \mathsf{resp}(\gamma) * \mathsf{state}(\gamma)$$

$$\mathsf{work}(\gamma) \stackrel{\text{def}}{=} \bigstar_{i \in \{i \mid \gamma^{\mathsf{s}}(i)=0\}} \big(\mathsf{work}[i] \mapsto 1\big) * \bigstar_{i \in \{i \mid \gamma^{\mathsf{s}}(i)\neq 0\}} \big(\mathsf{work}[i] \mapsto 0\big)$$

$$\mathsf{done}(\gamma) \stackrel{\text{def}}{=} \bigstar_{i \in \{i \mid \gamma^{\mathsf{s}}(i)=1\}} \big(\mathsf{done}[i] \mapsto 1\big) * \bigstar_{i \in \{i \mid \gamma^{\mathsf{s}}(i)\neq 1\}} \big(\mathsf{done}[i] \mapsto 0\big)$$

$$\mathsf{adj}(\gamma) \stackrel{\text{def}}{=} \bigstar_{i \in \gamma} \big(\bigstar_{j \in \gamma} \mathsf{a}[i][j] \mapsto \gamma^{\mathrm{E}}[i][j]\big) \qquad \mathsf{cost}(\gamma) \stackrel{\text{def}}{=} \bigstar_{i \in \gamma} \big(\mathsf{c}[i] \mapsto \gamma^{\mathsf{c}}(i)\big)$$

$$\mathsf{resp}(\gamma) \stackrel{\text{def}}{=} \bigstar_{i \in \gamma} \big(\bigstar_{j \in \gamma} r[i][j] \Rrightarrow \gamma^{\mathsf{r}}[i][j]\big) \qquad \mathsf{state}(\gamma) \stackrel{\text{def}}{=} \bigstar_{i \in \gamma} \big(s[i] \Rrightarrow \gamma^{\mathsf{s}}(i)\big)$$

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 `paralel_dijkstra`, before spawning new threads). We instantiate the capabilities of CoLoSL 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 has (potentially) 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 where mathematical actions are simply lifted to spatial ones indexed by the corresponding capability. Finally, the spatial postcondition Post is analogous to Pre and further states that all nodes in γ are processed (in `done`).

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

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

Verifying `parallel_dijkstra` A proof sketch of `dijkstra` is given in Fig. 6. 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 ($\mathsf{Inv}(\gamma, \mathtt{src}) \land \forall x \in \gamma.\, \gamma^{\mathsf{s}}(x) \doteq 1$) 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).

References

1. R. Bornat, C. Calcagno, and P. O'Hearn. Local reasoning, separation and aliasing. 2004.
2. T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms.* 2009.
3. E. Dijkstra, L. Lamport, A. Martin, C. Scholten, and E. Steffens. On-the-fly garbage collection: an exercise in cooperation. 1975.
4. T. Dinsdale-Young, L. Birkedal, P. Gardner, M. Parkinson, and H. Yang. Views: compositional reasoning for concurrent programs. 2013.
5. T. Dinsdale-Young, M. Dodds, P. Gardner, M. Parkinson, and V. Vafeiadis. Concurrent abstract predicates. 2010.
6. R. Dockins, A. Hobor, and A. Appel. A fresh look at separation algebras and share accounting. 2009.
7. X. Feng. Local rely-guarantee reasoning. 2009.
8. A. Grama, A. Gupta, G. Karypis, and V. Kumar. *Introduction to Parallel Computing.* 2003.
9. A. Hobor and J. Villard. The ramifications of sharing in data structures. 2013.
10. A. Nanevski, R. Ley-Wild, I. Sergey, and G. Delbianco. Communicating state transition systems for fine-grained concurrent resources. 2014.
11. A. Raad, A. Hobor, J. Villard, and P. Gardner. RGSep++: Encoding RGSep in the Views framework, 2015. `soundandcomplete.org/rgsep.pdf`.
12. A. Raad, J. Villard, and P. Gardner. CoLoSL: Concurrent Local Subjective Logic. 2015.
13. J. C. Reynolds. A short course on separation logic. 2003.
14. I. Sergey, A. Nanevski, and A. Banerjee. Mechanized verification of fine-grained concurrent programs. 2015.
15. K. Svendsen and L. Birkedal. Impredicative concurrent abstract predicates. 2014.
16. A. Turon, D. Dreyer, and L. Birkedal. Unifying refinement and hoare-style reasoning in a logic for higher-order concurrency. 2013.
17. H. Yang. *Local Reasoning for Stateful Programs.* PhD thesis, 2001.

1. $\{\mathsf{Pre}(t,\gamma_0)\}$
2. `void dijkstra(int[][] a, int[] c, int size, bitarray work, done){`
3. `i = 0;`
4. `while(done != 2^size-1){` $\left\{t * \left(\exists\gamma.\,\mathsf{g}(\gamma) * (\gamma_0 \doteq \gamma \wedge \mathsf{Inv}(\gamma,\mathsf{src}) \wedge \exists x.\,\gamma^{\mathsf{s}}(x) \neq 1)\right)_I\right\}$
5. `b = <CAS(work[i], 1, 0)>;` //perform A_t^1 if possible
6. `if(b){` $\left\{t * \left(\exists\gamma.\,\mathsf{g}(\gamma) * (\gamma_0 \doteq \gamma \wedge \mathsf{Inv}(\gamma,\mathsf{src}) \wedge \gamma^{\mathsf{s}}(i) = t \wedge \gamma^{\mathsf{r}}(i) = \bullet)\right)_I\right\}$
7. `cost = c[i];` $\left\{t * \left(\exists\gamma.\,\mathsf{g}(\gamma) * (\gamma_0 \doteq \gamma \wedge \mathsf{Inv}(\gamma,\mathsf{src}) \wedge \gamma^{\mathsf{s}}(i) = t \wedge \gamma^{\mathsf{r}}(i) = \bullet \wedge \mathtt{cost} = \gamma^{\mathsf{c}}(i))\right)_I\right\}$
8. `for(j=0;j<size;j++){` $\left\{t * \left(\exists\gamma.\,\mathsf{g}(\gamma) * (\gamma_0 \doteq \gamma \wedge \mathsf{Inv}(\gamma,\mathsf{src}) \wedge \gamma^{\mathsf{s}}(i) = t \wedge \gamma^{\mathsf{r}}(i) = 1^{\mathsf{j}}.0^{\mathsf{size-j}} \wedge \mathtt{cost} = \gamma^{\mathsf{c}}(i))\right)_I\right\}$
9. `newcost=cost+a[i][j]; b=true;` $\left\{t * \left(\begin{array}{l}\exists\gamma.\,\mathsf{g}(\gamma) * (\gamma_0 \doteq \gamma \wedge \mathsf{Inv}(\gamma,\mathsf{src}) \wedge \gamma^{\mathsf{s}}(i) = t \wedge \gamma^{\mathsf{r}}(i) = 1^{\mathsf{j}}.0^{\mathsf{size-j}} \\ \wedge\ \mathtt{cost} = \gamma^{\mathsf{c}}(i) \wedge \mathtt{newcost} = \mathtt{cost} + \gamma^{\mathsf{E}}[i][j] \wedge \mathtt{b} = 1\end{array}\right)_I\right\}$
10. `do{`
11. `oldcost=c[j];`//perform A_t^2 $\left\{t * \left(\begin{array}{l}\exists\gamma, c.\,\mathsf{g}(\gamma) * (\gamma_0 \doteq \gamma \wedge \mathsf{Inv}(\gamma,\mathsf{src}) \wedge \gamma^{\mathsf{s}}(i) = t \wedge \gamma^{\mathsf{r}}(i) = 1^{\mathsf{j}}.c.0^{\mathsf{size-j-1}} \\ \wedge \mathtt{cost} = \gamma^{\mathsf{c}}(i) \wedge \mathtt{newcost} = \mathtt{cost} + \gamma^{\mathsf{E}}[i][j] \wedge \mathtt{b} = 1 \wedge \mathtt{oldcost} = c)\end{array}\right)_I\right\}$
12. `if(newcost<oldcost){` $\left\{t * \left(\begin{array}{l}\exists\gamma, c.\,\mathsf{g}(\gamma) * (\gamma_0 \doteq \gamma \wedge \mathsf{Inv}(\gamma,\mathsf{src}) \wedge \gamma^{\mathsf{s}}(i) = t \wedge \gamma^{\mathsf{r}}(i) = 1^{\mathsf{j}}.c.0^{\mathsf{size-j-1}} \\ \wedge \mathtt{cost} = \gamma^{\mathsf{c}}(i) \wedge \mathtt{newcost} = \mathtt{cost} + \gamma^{\mathsf{E}}[i][j] \wedge \mathtt{oldcost} = c \wedge \mathtt{newcost} < \mathtt{oldcost})\end{array}\right)_I\right\}$
13. `b=<CAS(work[j],1,0)>;` //perform A_t^3 if possible
14. `if(b){` $\left\{t * \left(\begin{array}{l}\exists\gamma, c.\,\mathsf{g}(\gamma) * (\gamma_0 \doteq \gamma \wedge \mathsf{Inv}(\gamma,\mathsf{src}) \wedge \gamma^{\mathsf{s}}(i) = t \wedge \gamma^{\mathsf{r}}(i) = 1^{\mathsf{j}}.c.0^{\mathsf{size-j-1}} \wedge \mathtt{cost} = \gamma^{\mathsf{c}}(i) \\ \wedge \mathtt{newcost} = \mathtt{cost} + \gamma^{\mathsf{E}}[i][j] \wedge \mathtt{oldcost} = c \wedge \mathtt{newcost} < \mathtt{oldcost} \wedge \gamma^{\mathsf{s}}(j) = t \wedge \gamma^{\mathsf{r}}(j) = \circ)\end{array}\right)_I\right\}$
15. `b=<CAS(c[j],oldcost,newcost)>;` //perform A_t^4 if possible
$\left\{t * \left(\begin{array}{l}\exists\gamma, c.\,\mathsf{g}(\gamma) * (\gamma_0 \doteq \gamma \wedge \mathsf{Inv}(\gamma,\mathsf{src}) \wedge \gamma^{\mathsf{s}}(i) = t \wedge \left((\mathtt{b}=1 \wedge \gamma^{\mathsf{r}}(i) = 1^{\mathsf{j+1}}.0^{\mathsf{size-j-1}}) \vee (\mathtt{b}=0 \wedge \gamma^{\mathsf{r}}(i) = 1^{\mathsf{j}}.c.0^{\mathsf{size-j-1}})\right) \\ \wedge \mathtt{cost} = \gamma^{\mathsf{c}}(i) \wedge \mathtt{newcost} = \mathtt{cost} + \gamma^{\mathsf{E}}[i][j] \wedge \gamma^{\mathsf{s}}(j) = t \wedge \gamma^{\mathsf{r}}(j) = \circ)\end{array}\right)_I\right\}$
16. `<work[j] = 1>;` //perform A_t^5 or A_t^7 depending on the value of `b`
$\left\{t * \left(\begin{array}{l}\exists\gamma.\,\mathsf{g}(\gamma) * (\gamma_0 \doteq \gamma \wedge \mathsf{Inv}(\gamma,\mathsf{src}) \wedge \gamma^{\mathsf{s}}(i) = t \wedge \left((\mathtt{b}=1 \wedge \gamma^{\mathsf{r}}(i) = 1^{\mathsf{j+1}}.0^{\mathsf{size-j-1}}) \vee (\mathtt{b}=0 \wedge \gamma^{\mathsf{r}}(i) = 1^{\mathsf{j}}.0^{\mathsf{size-j}})\right) \\ \wedge \mathtt{cost} = \gamma^{\mathsf{c}}(i) \wedge \mathtt{newcost} = \mathtt{cost} + \gamma^{\mathsf{E}}[i][j])\end{array}\right)_I\right\}$
17. `}else{` $\left\{t * \left(\begin{array}{l}\exists\gamma, c.\,\mathsf{g}(\gamma) * (\gamma_0 \doteq \gamma \wedge \mathsf{Inv}(\gamma,\mathsf{src}) \wedge \gamma^{\mathsf{s}}(i) = t \wedge \gamma^{\mathsf{r}}(i) = 1^{\mathsf{j}}.c.0^{\mathsf{size-j-1}} \\ \wedge \mathtt{cost} = \gamma^{\mathsf{c}}(i) \wedge \mathtt{newcost} = \mathtt{cost} + \gamma^{\mathsf{E}}[i][j] \wedge \mathtt{oldcost} = c \wedge \mathtt{newcost} < \mathtt{oldcost})\end{array}\right)_I\right\}$
18. `b=<CAS(done[j],1,0)>;` //perform A_t^3 if possible
19. `if(b){` $\left\{t * \left(\begin{array}{l}\exists\gamma, c.\,\mathsf{g}(\gamma) * (\gamma_0 \doteq \gamma \wedge \mathsf{Inv}(\gamma,\mathsf{src}) \wedge \gamma^{\mathsf{s}}(i) = t \wedge \gamma^{\mathsf{r}}(i) = 1^{\mathsf{j}}.c.0^{\mathsf{size-j-1}} \wedge \mathtt{cost} = \gamma^{\mathsf{c}}(i) \\ \wedge \mathtt{newcost} = \mathtt{cost} + \gamma^{\mathsf{E}}[i][j] \wedge \mathtt{oldcost} = c \wedge \mathtt{newcost} < \mathtt{oldcost} \wedge \gamma^{\mathsf{s}}(j) = t \wedge \gamma^{\mathsf{r}}(j) = \bullet)\end{array}\right)_I\right\}$
20. `b=<CAS(c[j],oldcost,newcost)>;` //perform A_t^4 if possible
$\left\{t * \left(\begin{array}{l}\exists\gamma, c.\,\mathsf{g}(\gamma) * (\gamma_0 \doteq \gamma \wedge \mathsf{Inv}(\gamma,\mathsf{src}) \wedge \gamma^{\mathsf{s}}(i) = t \wedge \left((\mathtt{b}=1 \wedge \gamma^{\mathsf{r}}(i) = 1^{\mathsf{j+1}}.0^{\mathsf{size-j-1}}) \vee (\mathtt{b}=0 \wedge \gamma^{\mathsf{r}}(i) = 1^{\mathsf{j}}.c.0^{\mathsf{size-j-1}})\right) \\ \wedge \mathtt{cost} = \gamma^{\mathsf{c}}(i) \wedge \mathtt{newcost} = \mathtt{cost} + \gamma^{\mathsf{E}}[i][j] \wedge \gamma^{\mathsf{s}}(j) = t \wedge \gamma^{\mathsf{r}}(j) = \bullet)\end{array}\right)_I\right\}$
21. `if(b){<work[j]=1>}else{<done[j]=1>}` $//A_t^6$ or A_t^7 based on `b`
$\left\{t * \left(\begin{array}{l}\exists\gamma.\,\mathsf{g}(\gamma) * (\gamma_0 \doteq \gamma \wedge \mathsf{Inv}(\gamma,\mathsf{src}) \wedge \gamma^{\mathsf{s}}(i) = t \wedge \left((\mathtt{b}=1 \wedge \gamma^{\mathsf{r}}(i) = 1^{\mathsf{j+1}}.0^{\mathsf{size-j-1}}) \vee (\mathtt{b}=0 \wedge \gamma^{\mathsf{r}}(i) = 1^{\mathsf{j}}.0^{\mathsf{size-j}})\right) \\ \wedge \mathtt{cost} = \gamma^{\mathsf{c}}(i) \wedge \mathtt{newcost} = \mathtt{cost} + \gamma^{\mathsf{E}}[i][j])\end{array}\right)_I\right\}$
22. `}}}` $\left\{t * \left(\begin{array}{l}\exists\gamma.\,\mathsf{g}(\gamma) * (\gamma_0 \doteq \gamma \wedge \mathsf{Inv}(\gamma,\mathsf{src}) \wedge \gamma^{\mathsf{s}}(i) = t \wedge \mathtt{cost} = \gamma^{\mathsf{c}}(i) \\ \left[(\mathtt{newcost} < \mathtt{oldcost} \wedge ((\mathtt{b}=1 \wedge \gamma^{\mathsf{r}}(i) = 1^{\mathsf{j+1}}.0^{\mathsf{size-j-1}}) \vee (\mathtt{b}=0 \wedge \gamma^{\mathsf{r}}(i) = 1^{\mathsf{j}}.0^{\mathsf{size-j}}))) \right. \\ \left. \vee (\mathtt{newcost} \geqslant \mathtt{oldcost} \wedge \mathtt{b}=1 \wedge \gamma^{\mathsf{r}}(i) = 1^{\mathsf{j}}.-.0^{\mathsf{size-j-1}})\right]\end{array}\right)_I\right\}$
//perform A_t^8 for 3rd disjunct $\left\{t * \left(\begin{array}{l}\exists\gamma.\,\mathsf{g}(\gamma) * (\gamma_0 \doteq \gamma \wedge \mathsf{Inv}(\gamma,\mathsf{src}) \wedge \gamma^{\mathsf{s}}(i) = t \wedge \mathtt{cost} = \gamma^{\mathsf{c}}(i) \\ \left((\mathtt{b}=1 \wedge \gamma^{\mathsf{r}}(i) = 1^{\mathsf{j+1}}.0^{\mathsf{size-j-1}}) \vee (\mathtt{b}=0 \wedge \gamma^{\mathsf{r}}(i) = 1^{\mathsf{j}}.0^{\mathsf{size-j}})\right)\end{array}\right)_I\right\}$
23. `} while(!b)` $\left\{t * \left(\exists\gamma.\,\mathsf{g}(\gamma) * (\gamma_0 \doteq \gamma \wedge \mathsf{Inv}(\gamma,\mathsf{src}) \wedge \gamma^{\mathsf{s}}(i) = t \wedge \gamma^{\mathsf{r}}(i) = 1^{\mathsf{j+1}}.0^{\mathsf{size-j-1}} \wedge \mathtt{cost} = \gamma^{\mathsf{c}}(i))\right)_I\right\}$
24. `}` $\left\{t * \left(\exists\gamma.\,\mathsf{g}(\gamma) * (\gamma_0 \doteq \gamma \wedge \mathsf{Inv}(\gamma,\mathsf{src}) \wedge \gamma^{\mathsf{s}}(i) = t \wedge \gamma^{\mathsf{r}}(i) = \bullet)\right)_I\right\}$
25. `<done[i]=1>;` //perform A_t^9 $\left\{t * \left(\exists\gamma.\,\mathsf{g}(\gamma) * (\gamma_0 \doteq \gamma \wedge \mathsf{Inv}(\gamma,\mathsf{src}) \wedge \gamma^{\mathsf{s}}(i) = 1)\right)_I\right\}$
26. `} i = (i+1) mod size; }` $\left\{t * \left(\exists\gamma.\,\mathsf{g}(\gamma) * (\gamma_0 \doteq \gamma \wedge \mathsf{Inv}(\gamma,\mathsf{src}) \wedge \forall x.\,\gamma^{\mathsf{s}}(x) = 1)\right)_I\right\}$
27. `}` $\{\mathsf{Post}(t,\gamma_0)\}$

Figure 6: Proof sketch of the speculative `dijkstra` algorithm.

A Appendix

A.1 Marking a Graph in Parallel

In this section we use our techniques to verify a parallel algorithm that marks every node in a heap-represented graph. We choose parallel marking as our first target because it is the simplest member of an important class of related parallel algorithms that includes map, fold/reduce, and spanning/dispose.

Consider the following two-processor program `pmark` that marks every reachable node in a graph starting from root `v`:

```
1  type node {bool mark; node* left, right;};
2
3  void cmark(node* x) {
4    node* l, r; bool flag;
5    if (x == 0) return;
6    <if (x->mark == ○) {x->mark = •; flag = true;} else flag = false>;
7    if (flag) {
8      l = x->left; r = x->right;
9      cmark(l); cmark(r);
10   }
11 }
12
13 void pmark(node* x) {
14   node* l, r;
15   if (x == 0) return;
16   l = x->left; r = x->right;
17   if (x->mark == ○) x->mark = • else return;
18   cmark(l) || cmark(r);
19 }
```

Line 1 gives the data type of heap-represented graphs. Lines 4–11 give the recursive concurrent marking subalgorithm `cmark`, two copies of which will be run in parallel. Line 6 uses our pseudo-language's atomic instruction construct `<...>`, which guarantees that the enclosed instructions cannot be interrupted by other threads. Note that in in this case the atomic operation is the Compare and Swap (CAS) primitive. Line 6 uses ○ for `0`/`false` and • for `1`/`true` when interacting with the `mark` bit to increase readability, a pattern we continue hereafter. With the exception of the CAS, `cmark` is just the standard sequential depth-first marking algorithm. Lines 14–19 contain the parallel marking algorithm `pmark`, which is also very similar to the standard sequential algorithm. The critical point is line 19, which sets up the parallel computation using our pseudolanguage's `C1 || C2` construction, which indicates standard fork-join parallelism. Here, we consider only a single pair of parallel threads fto simplify the mathematical argument. It is straightforward to extend to more threads as in the algorithm of `copy_dag` in Fig. 3 (§3).

The correctness argument of `cmark` is rather subtle due to the deep unspecified sharing inside the graph as well as the parallel behaviour of the algorithm.

This subtlety is not too surprising since the unspecified sharing makes verifying even the sequential version of `cmark` not entirely trivial [9]. However, the non-deterministic behaviour of parallel computation makes even *specifying* the behaviour of `cmark` rather challenging.

Mathematical graphs A mathematical graph $\gamma \in \mathcal{W}_\gamma$ is a triple (V, E, L), where V is the vertex set; $E : V \to (V_0 \times V_0)$ is the edge function with $V_0 \stackrel{\text{def}}{=} V \uplus \{0\}$, where 0 denotes the absence of an edge; and $L : V \to D$ is the vertex labelling function with $D \stackrel{\text{def}}{=} \{\circ, \bullet\} \times (\mathbb{N}\times\mathbb{N})$. The first component describes whether a node is yet to be marked ($\circ$), or it has already been marked ($\bullet$). The second component denotes the *promise* set described shortly.
As before, we write γ^{V}, γ^{E} and γ^{L} to project out the various components of γ; we write $\gamma^{\mathsf{l}}(x)$ and $\gamma^{\mathsf{r}}(x)$ for the first and second projections of $E(x)$; and write $\gamma(x)$ for $(L(x), \gamma^{\mathsf{l}}(x), \gamma^{\mathsf{r}}(x))$ when $x \in V$. We further write $\gamma^{\mathsf{m}}(x)$ and $\gamma^{\mathsf{p}}(x)$ for the first and second projections of $\gamma^{\mathsf{L}}(x)$. Congruence is defined analogously when two graphs γ_1 and γ_2 have the same vertices and edges: $\gamma_1 \cong \gamma_2 \stackrel{\text{def}}{=} \gamma_1^{\mathsf{V}}{=}\gamma_2^{\mathsf{V}} \wedge \gamma_1^{\mathsf{E}}{=}\gamma_2^{\mathsf{E}}$.
We define the *path* relation, $\overset{\gamma}{\rightsquigarrow}$, and its reflexive transitive closure, $\overset{\gamma}{\rightsquigarrow}{}^*$, as:

$$x \overset{\gamma}{\rightsquigarrow} y \stackrel{\text{def}}{=} \gamma(x)^{\mathsf{l}}{=}y \vee \gamma(x)^{\mathsf{r}}{=}y \qquad x \overset{\gamma}{\rightsquigarrow}{}^* y \stackrel{\text{def}}{=} x{=}y \vee \exists z.\, x \overset{\gamma}{\rightsquigarrow} z \wedge z \overset{\gamma}{\rightsquigarrow}{}^* y$$

We also define the *white path* relation, $\overset{\gamma}{\rightsquigarrow}_\circ$, and its reflexive transitive closure, $\overset{\gamma}{\rightsquigarrow}{}^*_\circ$, as:

$$x \overset{\gamma}{\rightsquigarrow}_\circ y \stackrel{\text{def}}{=} x \overset{\gamma}{\rightsquigarrow} y \wedge \gamma^{\mathsf{m}}(x){=}\circ \wedge \gamma^{\mathsf{m}}(y){=}\circ$$
$$x \overset{\gamma}{\rightsquigarrow}{}^*_\circ y \stackrel{\text{def}}{=} (x{=}y \wedge \gamma^{\mathsf{m}}(x){=}\circ) \vee (\exists z.\, x \overset{\gamma}{\rightsquigarrow}_\circ z \wedge z \overset{\gamma}{\rightsquigarrow}{}^*_\circ y)$$

Tokens As before, to track the contribution of each thread and record the overall progress of the program we appeal to a token mechanism. In comparison to the `copy_dag` example in §3, our token mechanism is much simpler since the number of threads running in parallel is fixed (two) and decided at the beginning (in `cmark`). We define our tokens as elements of the set $\mathcal{T} \stackrel{\text{def}}{=} \{\mathsf{B}, \mathsf{R}\}$ denoting the B(lue) and R(ed) threads, respectively. Observe that we can generalise this token mechanism to arbitrarily-many tokens as in §3.

Let us turn to the specification of the concurrent marking function `cmark`. Consider the diagrammed "execution trace" in Fig. 7, in which two threads, B(lue) and R(ed), are cooperating to mark the diagrammed graph. We indicate that the t-colored thread is "currently processing" `cmark(`x`)` by putting t in a star near node x.

Initially (i), the graph is unmarked and both threads are working on different nodes. After marking their initial node, the threads compete to mark the middle node in (ii). In (iii), thread B has won the race; accordingly, thread R is done and returns. The problem is: how do we specify the postcondition of thread R? Clearly it has done something vital: without it, the graph would not be completely marked once thread B is finished. However, thread R returns even though the last node (which was reachable along a white path from its initial node in the original graph) is not yet marked in (iii). That is, the postcondition of thread R cannot promise that the final node is marked when it returns. Dually, the postcondition of thread B also cannot promise that the final node is marked

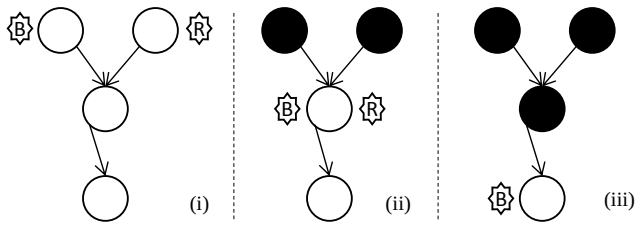


Figure 7: An execution trace of two parallel `cmark`s.

when it returns (since thread R might have won the race). This leaves us with a nasty problem when we combine the postconditions of both parallel calls in line 19: since **neither** thread guarantees that all of the nodes reachable from their single inputs (`l` and `r`, respectively) are marked, it seems rather difficult to prove that afterwards, **all** nodes reachable from **both** inputs are marked!

Informally, the reason that thread R can safely return in (ii) is that when it is blocked from reaching a node x, then thread B will mark x before returning. In other words, there is a "promise" that is transferred: when thread R reaches a marked node x, it assumes that "whoever" marked x is responsible for marking the children of x. In the case of Fig. 7, thread B marked the middle node and is thus responsible for the last one. There are other similar situations that involve this kind of obligation handoff. For example, when the graph has cycles, a thread may encounter a node that *it marked itself in a previous recursive call*; the obligation in this case is a promise to visit any remaining children as the recursive call stack unwinds (by which time, of course, said children may have been marked by other threads, leading to further promises).

To represent these promises, we appeal to our token mechanism. We associate each node in the graph with a *promise* set defined as a *multiset* of tokens describing how many times each thread is obliged to visit it. For instance, for a node x with promise set $\{\mathsf{B},\mathsf{B},\mathsf{R},\mathsf{R},\mathsf{R}\}$, thread B is obliged to visit x two times (i.e. via two distinct paths), while thread R is obliged to visit x three times (i.e. via three distinct paths). We model this by defining the second component of a node labels (recall that $D \stackrel{\text{def}}{=} \{\circ,\bullet\} \times (\mathbb{N}\times\mathbb{N})$) as a pair of non-negative integers $(\sharp\mathsf{B},\sharp\mathsf{R})$, recording the number of times each thread is to visit it. For brevity, given $t \in \mathcal{T}=\{\mathsf{B},\mathsf{R}\}$, we write $\bar{t}$ for the "other color", e.g. $\overline{\mathsf{R}} = \mathsf{B}$. Moreover, we write $\gamma^{\mathsf{p}}(x)[t]$ to indicate the number of t-coloured tokens on node x; we also write $\gamma[x.t\text{--}]$ and $\gamma[x.t\text{++}]$ to denote the graph obtained after decrementing and increasing the number of t tokens on node x in γ, respectively.

Actions We define two families of actions, each of which indexed by a token t. The first set of actions, A^1_t, describes the `CAS` operation of thread t in line 6 when successful. That is, when visiting node x, thread t discovers that x is currently unmarked. It then marks x, removes one t token from the promise set of x and simultaneously adds a t tokens to each of x's immediate children.

$$A_t^1 \stackrel{\text{def}}{=} \left\{ (\gamma,\gamma') \middle| \begin{array}{l} \gamma^{\mathsf{m}}(x){=}\circ \wedge \gamma^{\mathsf{p}}(x)[t] > 0 \wedge E(x){=}(l,r) \\ \wedge\ \exists \gamma''.\, \gamma''{=}(\gamma^{\mathsf{V}}, \gamma^{\mathsf{E}}, \gamma^{\mathsf{L}}[x \mapsto (\bullet, \gamma^{\mathsf{p}}(x))] \\ \wedge\ \gamma'{=}\gamma''[x.t\text{--}][l.t{+}{+}][r.t{+}{+}] \end{array} \right\}$$

Similarly, the next set of actions captures the failure case of the `CAS` operation of thread t in line 6. That is, when visiting node x, thread t discovers that x is already marked. It then marks x, removes one t token from the promise set of x and simultaneously adds a t tokens to each of x's immediate children. It then proceeds by (ghostly) removing its token from x.

$$A_t^2 \stackrel{\text{def}}{=} \left\{ (\gamma,\gamma') \middle| \gamma^{\mathsf{m}}(x){=}\bullet \wedge \gamma^{\mathsf{p}}(x)[t] > 0 \wedge \gamma'{=}\gamma[x.t\text{--}] \right\}$$

We write A_t for actions of thread t: $A_t \stackrel{\text{def}}{=} A_t^1 \cup A_t^2$.

Mathematical specification Suppose a node x in γ has a token t on it, and consider an arbitrary node y. The key insight is that thread t's promise to do some "work" on x also extends, somewhat loosely, to y if y is reachable from x along a white path. The "loose" qualifier is because more than one token/thread can be responsible for y, if more than one token can reach y via a white path. We capture this by the "responsibility set" of y, written $\mathcal{R}(\gamma, y)$, denoting the set of tokens that can reach y via a white path. That is, those threads that are jointly (and severally) responsible for processing y:

$$\mathcal{R}(\gamma, y) \stackrel{\text{def}}{=} \{t \mid \exists x.\, \gamma^{\mathsf{p}}(x)[t] > 0 \wedge x \overset{\gamma}{\rightsquigarrow}{}^{*}_{\circ}\, y\}$$

Observe that first, responsibility sets represent the "worst case" liability; second, responsibility sets monotonically shrink: as threads fulfil their obligations, they sometimes take individual responsibility for some of the remaining tasks; and third (and most crucially) when the responsibility set of a node is empty, then it must be marked.[2] This last observation is the crux of the invariant for `cmark`. Throughout the execution of `cmark`, the graph γ satisfies the invariant $\mathsf{Inv}(\gamma)$ described below.

$$\mathsf{Inv}(\gamma) \stackrel{\text{def}}{=} \forall x \in \gamma.\, \mathcal{R}(\gamma, x){=}\varnothing \implies \gamma^{\mathsf{m}}(x){=}\bullet$$

Note that for all threads t, the invariant $\mathsf{Inv}(\gamma)$ is stable with respect to the actions of t (i.e. A_t). More concretely, when updating a node $x \in \gamma$ via the actions of A_t, for an arbitrary node $y \neq x$, if y has a non-empty responsibility set before the update, it will still have a non-empty responsibility set afterwards. That is, if $t \in \mathcal{R}(\gamma, y)$ before the update, then $t \in \mathcal{R}(\gamma, y)$ afterwards. Dually, if $t \notin \mathcal{R}(\gamma, y)$ before the update, then $\mathcal{R}(\gamma, y)$ remains unchanged and thus $t \notin \mathcal{R}(\gamma, y)$ afterwards. No vertex is left behind: the only way the responsibility set for y can become empty is when $x = y$, i.e. when y has just been updated. This is captured by Lemma 3 below.

The precondition of `cmark(`x`)`, $\mathsf{P}_1^t(x, \gamma)$, is as defined below where x identifies the top node being marked, t is the thread identifier, and γ denotes the original

[2] We refer the reader to the end of this section (page 26, paragraph "Example: responsibility sets and tokens") for a detailed example of responsibility sets and tokens.

graph. It asserts that there is at least one t token on x (when $\neq 0$).

$$\mathsf{P}_1^t(x,\gamma) \stackrel{\text{def}}{=} (x \neq 0 \Rightarrow \gamma^{\mathsf{p}}(x)[t] > 0$$

The postcondition of `cmark(`x`)`, $\mathsf{Q}_1^t(x,\gamma,\gamma')$, is as defined below where x identifies the top node being marked, t is the thread identifier, γ and γ' respectively denote the original and the final resultant graph. It asserts that the t tokens of all nodes but x remain unchanged and that the t tokens of x is decremented by one.

$$\mathsf{Q}_1^t(x,\gamma,\gamma') \stackrel{\text{def}}{=} (x \neq 0 \Rightarrow \gamma[x.t\text{--}] \stackrel{t}{=} \gamma')$$

where
$$\gamma \stackrel{t}{=} \gamma' \stackrel{\text{def}}{=} \forall x \in \gamma.\, \gamma^{\mathsf{p}}(x)[t]{=}\gamma'^{\mathsf{p}}(x)[t]\gamma'^{\mathsf{p}}(x)[t]\big)$$

Observe that for a given thread t, the pre- and post-conditions of `cmark` are stable with respect to the actions of the other thread, i.e. $A_{\bar{t}}$. This is captured in Lemma 3.

The precondition of `pmark(`x`)`, $\mathsf{P}_2(x,\gamma)$, is as defined below where x identifies the top node being marked, and γ denotes the graph. It asserts that x (when $\neq 0$) is a node in γ (first conjunct), and that all nodes in γ are unmarked and are reachable from x (second conjunct). The latter part is described via the white path relation, $\stackrel{\gamma}{\rightsquigarrow}{}^*_\circ$, described earlier.

$$\mathsf{P}_2(x,\gamma) \stackrel{\text{def}}{=} (x \neq 0 \Rightarrow x \in \gamma^{\mathsf{v}}) \wedge \forall y \in \gamma^{\mathsf{v}}.\, x \stackrel{\gamma}{\rightsquigarrow}{}^*_\circ y$$

Finally, the postcondition of `pmark(`x`)`, $\mathsf{Q}_2(x,\gamma)$, is as defined below. It asserts that all vertices in γ are marked.

$$\mathsf{Q}_2(x,\gamma) \stackrel{\text{def}}{=} \forall y \in \gamma^{\mathsf{v}}.\, \gamma^{\mathsf{m}}(y){=}\bullet$$

Lemma 3. *For all mathematical graphs $\gamma, \gamma', \gamma''$ and all tokens t,*

$$\mathsf{Inv}(\gamma) \wedge \gamma A_t \gamma' \Rightarrow \mathsf{Inv}(\gamma')$$
$$\mathsf{P}_1^t(x,\gamma) \wedge \gamma A_{\bar{t}} \gamma' \Rightarrow \mathsf{P}_1^t(x,\gamma')$$
$$\mathsf{Q}_1^t(x,\gamma,\gamma') \wedge \gamma' A_{\bar{t}} \gamma'' \Rightarrow \mathsf{Q}_1^t(x,\gamma,\gamma'')$$

Proof. Follows from the definitions of A_t, Inv, P_1, and Q_1.

Spatial representation We represent a mathematical graph γ in the heap, through the predicate, as a linked structure :

$$\mathsf{graph}(x,\gamma) \stackrel{\text{def}}{=} (x = 0 \wedge \mathsf{emp}) \vee \big(\mathsf{node}(x,\gamma) \circledast \mathsf{graph}(\gamma^{\mathsf{l}}(x),\gamma) \circledast \mathsf{graph}(\gamma^{\mathsf{r}}(x),\gamma)\big)$$
$$\mathsf{node}(x,\gamma) \stackrel{\text{def}}{=} \exists m, t_1, t_2, l, r.\, \gamma(x) = \big((m,(t_1,t_2)),l,r\big) \wedge x \mapsto m,l,r * x \Rightarrow t_1,t_2$$

Here, $\circledast$ is the "overlapping conjunction" [9]:

$$h \models_{\mathsf{SL}} P \circledast Q \stackrel{\text{def}}{=} \exists h_1,h_2,h_3.\, (h_1 \oplus h_2 \oplus h_3 = h) \wedge (h_1 \oplus h_2 \vDash P) \wedge (h_2 \oplus h_3 \vDash Q)$$

where $\oplus$ denotes heap composition (usually disjoint union). That is, graph is a recursive predicate such that when applied to 0, it is emp, and applied to a

nonzero address x, it is composed of three potentially overlapping parts: the left and right subgraphs of x, and $\mathsf{node}(x)$, implementing the node x. The, $\mathsf{node}(x,\gamma)$ implements a node x in γ via the basic separation logic "maps-to" predicate $\mapsto$ to store the mark bit and left/right subgraphs in adjacent cells in the heap. The additional information $(t_1,t_2) \in \mathcal{T}$ is implemented in the "ghost heap": it is used in the proof but not actually modified by the program. We use $\Rightarrow$ to differentiate the ghost heap from the concrete heap indicated by $\mapsto$.
We can now specify the pre- and post-conditions of `cmark` as follows:

$$\mathsf{Pre}_1^t(x,\gamma_0) \stackrel{\text{def}}{=} t * \boxed{\exists\gamma.\, g \Rightarrow \gamma * (\gamma_0 \dot{\cong} \gamma \wedge \mathsf{Inv}(\gamma) \wedge \mathsf{P}_1^t(x,\gamma)) * \mathsf{graph}(x,\gamma)}_I$$

$$\mathsf{Post}_1^t(x,\gamma_0) \stackrel{\text{def}}{=} t * \boxed{\exists\gamma.\, g \Rightarrow \gamma * (\gamma_0 \dot{\cong} \gamma \wedge \mathsf{Inv}(\gamma) \wedge \mathsf{Q}_1^t(x,\gamma_0,\gamma)) * \mathsf{graph}(x,\gamma)}_I$$

$$I \stackrel{\text{def}}{=} \left\{ t : \mathsf{graph}(x,\gamma) \wedge \gamma A_t \gamma' \rightsquigarrow \mathsf{graph}(x,\gamma') \right.$$

Verifying `cmark` A proof sketch of `cmark` is given in Fig. 8 . One thing jumps out when looking at the assertions at each program point: they have **identical** spatial parts:

$$t * \boxed{\exists\gamma.\, g \Rightarrow \gamma * \mathsf{graph}(\mathsf{x},\gamma) * \ldots}_I$$

Of course, the heap is changing constantly, due both to the actions of this thread as well as the other. The spatial part is saying that the heap is staying in sync with the mathematical graph γ, however γ may be changing. Whenever this thread interacts with the shared state, the mathematical graph γ will change; changes to γ due to the other threads are handled by the existential quantifier.

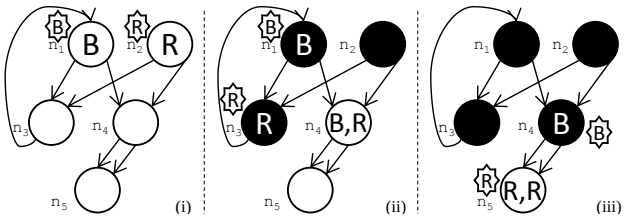
What does change at each program point are the pure facts. Line 1 contains the precondition for `cmark`. On line 2 we check to see if `x` is 0. If so the program returns and the postcondition, $\mathsf{Post}_1^t(\mathsf{x},\gamma_0)$, follows trivially from the definition of the precondition $\mathsf{Pre}_1^t(\mathsf{x},\gamma_0)$. If $\mathsf{x} \neq 0$, then the atomic block of line 6 is executed. We first check if `x` unmarked; if so we perform the action A_t^1. In doing so, we set the flag (`flag`) to true, remove a token t from `x`, and add a token t to each of the children of `x`. On the other hand, if `x` is already marked, we perform the action A_t^2. We thus set the flag (`flag`) to false, and remove a token t from `x` and thus arrive at the desired post-condition $\mathsf{Post}_1^t(\mathsf{x},\gamma_0)$.

On line 7 we check whether the thread did mark `x` and has thus incurred an obligation to call `cmark` on `x`'s children. If this is the case, we load the left and right children of `x` into `l` and `r` (line 8) and subsequently call `cmark` on them (lines 9 and 10).

Verifying `pmark` Examine the parallel calls to `cmark` in the `pmark` function (line 19). To set up the parallel behaviour, the `pmark` ensures (using ghost state) that there is one B token on the initial vertex `x`'s left child, one R token on `x`'s right child, and no other tokens elsewhere. This, plus marking `x` itself in line 17, ensures the precondition of `cmark`, $\mathsf{Pre}_1^t(\mathsf{x},\gamma_0)$, for each thread t (in particular the key invariant Inv and the precondition $\mathsf{P}_1^t(\mathsf{x},\gamma_0)$). Since each parallel `cmark` call operates on only one kind of token, we have a kind of disjointness to ensure

Figure 8: Proof sketch of the cmark program for thread t.

compositional reasoning during the parallel calls. After the calls return, we know (from the postcondition of cmark on the left branch) that there are no B tokens anywhere, and similarly there are no R tokens anywhere, allowing us to use the



$$\mathcal{R}(\gamma, v) = \begin{cases} & \mathtt{n_1} & \mathtt{n_2} & \mathtt{n_3} & \mathtt{n_4} & \mathtt{n_5} \\ \text{i} & \{\mathsf{B},\mathsf{R}\} & \{\mathsf{R}\} & \{\mathsf{B},\mathsf{R}\} & \{\mathsf{B},\mathsf{R}\} & \{\mathsf{B},\mathsf{R}\} \\ \text{ii} & \varnothing & \varnothing & \varnothing & \{\mathsf{B},\mathsf{R}\} & \{\mathsf{B},\mathsf{R}\} \\ \text{iii} & \varnothing & \varnothing & \varnothing & \varnothing & \{\mathsf{R}\} \end{cases}$$

Figure 9: Execution snapshots of `cmark` with responsibility tokens.

invariant Inv, to prove that the entire graph has been marked and thus to satisfy the postcondition of `pmark`, $\mathsf{Post}_2(\mathsf{x}, \gamma_0)$.

Example: responsibility sets and tokens We can see this protocol in action in Fig. 9, which shows three snapshots of the algorithm and calculates the responsibility sets $\mathcal{R}(\gamma, v)$ for each vertex v at the graph γ depicted in each snapshot. Although we only have five nodes ($\mathtt{n_1}$–$\mathtt{n_5}$), the input is nontrivial, with cycles, deep sharing, and several potential races. Letter t in a star indicates where thread t is currently working. We put letters inside the vertices to indicate the token counts. Note that to get from (i) to (ii), the B thread has gone around the $\mathtt{n_1}$-$\mathtt{n_3}$-$\mathtt{n_1}$ loop, first taking off its token as it colors $\mathtt{n_1}$ and then later putting its token back on $\mathtt{n_1}$ as it colors $\mathtt{n_3}$ (after winning the race); in (ii) both threads are about to notice that the vertices they are looking at are already colored.

Notice that responsibility sets represent "worst case" liability; in (i) it is unlikely that R will mark $\mathtt{n_2}$, $\mathtt{n_3}$, and $\mathtt{n_1}$ before B does any work, but it is possible if the scheduler is particularly unkind; in contrast, there is no way B will ever mark $\mathtt{n_2}$. Second, responsibility sets monotonically shrink; as threads fulfill their obligations, they sometimes take individual responsibility for some of the remaining tasks, as in (iii) with vertex $\mathtt{n_5}$. Finally, and most crucially: when a vertex's responsibility set is empty then that vertex must be marked.

A.2 Computing the Spanning Tree of a Graph in Parallel

Mathematical graphs A mathematical graph, $\gamma \in \Gamma$, 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 (a null pointer); and $L = V \to D$ is the label function.

Observe that as with `copy_dag` in §3, the `span` program spawns a new thread at each recursive call point in line 8. We thus take our tokens as elements of the tree share algebra, $\pi \in \Pi$, described in §3. We define our labels as $D \stackrel{\text{def}}{=} \{0\} \uplus \Pi$. That is, for each node the label function records whether it is yet to be visited (0), or it has been already visited and marked by a thread π (π). Given a graph $\gamma = (V, E, L)$ we write γ^{V} for V, γ^{E} for E, and γ^{L} for L. Moreover, we write $\gamma^{\mathsf{l}}(x) = l$ and $\gamma^{\mathsf{r}}(x) = r$) when $\gamma^{\mathsf{E}}(x) = (l, r)$, and write $\gamma^{\mathsf{m}}(x)$ for $\gamma^{\mathsf{L}}(x)$. As before, we define the path relation ($\overset{\gamma}{\rightsquigarrow}$), and its reflexive transitive closure as:

$$x \overset{\gamma}{\rightsquigarrow} y \stackrel{\text{def}}{=} \gamma^{\mathsf{l}}(x) = y \vee \gamma^{\mathsf{r}}(x) = y \qquad x \overset{\gamma}{\rightsquigarrow}{}^{*} y \stackrel{\text{def}}{=} x = y \vee (\exists z.\, x \overset{\gamma}{\rightsquigarrow}{}^{*} z \wedge z \overset{\gamma}{\rightsquigarrow} y)$$

Actions We define two families of actions, each of which indexed by a token $\pi \in \Pi$. The first set, A^1_π, describes the action of line 6 in the algorithm: the state of a node is changed from unmarked to being processed by thread π.

$$A^1_\pi \stackrel{\text{def}}{=} \left\{ \big((V, E, L), (V, E, L')\big) \mid L(x) = 0 \wedge L' = [x \mapsto \pi] L \right\}$$

The next two set of actions respectively describe the operations of lines 9 and 10.

$$A^2_\pi \stackrel{\text{def}}{=} \left\{ \big((V,E,L), (V,E',L)\big) \,\middle|\, \begin{array}{l} L(x) = \pi \wedge E(x) = (l, r) \wedge L(l) = \pi' \wedge \pi' \neq \pi.\mathsf{l} \\ \wedge\ E' = [x \mapsto (0, r)] E \end{array} \right\}$$

$$A^3_\pi \stackrel{\text{def}}{=} \left\{ \big((V,E,L), (V,E',L)\big) \,\middle|\, \begin{array}{l} L(x) = \pi \wedge E(x) = (l, r) \wedge L(r) = \pi' \wedge \pi' \neq \pi.\mathsf{r} \\ \wedge\ E' = [x \mapsto (l, 0)] E \end{array} \right\}$$

We write A_π for actions of thread π: $A_\pi \stackrel{\text{def}}{=} A^1_\pi \cup A^2_\pi \cup A^3_\pi$.

Mathematical specification

$$\begin{array}{l}
\mathsf{Inv}(\gamma, t) \stackrel{\text{def}}{=} \forall x, \pi.\, \gamma^{\mathsf{m}}(x) = \pi \Rightarrow t \overset{\gamma}{\rightsquigarrow}_\pi x \\
t \overset{\gamma}{\rightsquigarrow}_\pi x \stackrel{\text{def}}{=} \gamma^{\mathsf{m}}(x) = \pi \wedge t \overset{\gamma}{\rightsquigarrow}\!\!\!|_\pi x \\
t \overset{\gamma}{\rightsquigarrow}\!\!\!|_\pi x \stackrel{\text{def}}{=} x = t \vee \exists y, \pi'.\, t \overset{\gamma}{\rightsquigarrow}_{\pi'} y \wedge \big((\pi = \pi'.\mathsf{l} \wedge \gamma^{\mathsf{l}}(y) = x) \vee (\pi = \pi'.\mathsf{r} \wedge \gamma^{\mathsf{r}}(y) = x)\big) \\
\gamma \leqslant_t \gamma_0 \stackrel{\text{def}}{=} \gamma^{\mathsf{V}} = \gamma_0^{\mathsf{V}} \wedge \gamma^{\mathsf{E}} \subseteq \gamma_0^{\mathsf{E}} \wedge \forall x.\, t \overset{\gamma_0}{\rightsquigarrow}{}^{*} x \Rightarrow t \overset{\gamma}{\rightsquigarrow}{}^{*} x \\
\mathsf{P}^\pi(\gamma, t, x) \stackrel{\text{def}}{=} \forall x, \pi'.\, \gamma^{\mathsf{m}}(x) = \pi' \Rightarrow \pi' \sharp \pi \wedge (x = 0 \vee t \overset{\gamma}{\rightsquigarrow}\!\!\!|_\pi x) \\
\mathsf{Q}^\pi_{\mathsf{s}}(\gamma, x) \stackrel{\text{def}}{=} (x = 0 \vee \gamma^{\mathsf{m}}(x) = \pi) \\
\mathsf{Q}^\pi_{\mathsf{f}}(\gamma, x) \stackrel{\text{def}}{=} \exists \pi'.\, \pi' \sharp \pi \wedge \gamma^{\mathsf{m}}(x) = \pi'
\end{array}$$

Lemma 4. *For all mathematical graphs γ and γ', nodes t and x, and tokens π and π',*

$$\mathsf{Inv}(\gamma,t) \wedge \gamma\, A_\pi\, \gamma' \Rightarrow \mathsf{Inv}(\gamma',t) \tag{4}$$

$$\mathsf{P}^{\pi'}(\gamma,t,x) \wedge \pi \notin \overline{\pi'} \wedge \gamma\, A_\pi\, \gamma' \Rightarrow \mathsf{P}^{\pi'}(\gamma',t,x) \tag{5}$$

$$\mathsf{Q}_\mathsf{s}^{\pi'}(\gamma,x) \wedge \gamma\, A_\pi\, \gamma' \Rightarrow \mathsf{Q}_\mathsf{s}^{\pi'}(\gamma',x) \tag{6}$$

$$\mathsf{Q}_\mathsf{f}^{\pi'}(\gamma,x) \wedge \gamma\, A_\pi\, \gamma' \Rightarrow \mathsf{Q}_\mathsf{f}^{\pi'}(\gamma',x) \tag{7}$$

Proof. Follows from the definitions of A_π, Inv, P, Q_s, and Q_f.

Part (4) states that the invariant Inv is stable with respect to the actions of all threads (A_π for any tokens π). Part (5) states that the precondition of thread π' ($\mathsf{P}^{\pi'}$) is stable with respect to the actions of all threads but those of its descendants ($\pi \notin \overline{\pi'}$). Parts (6) and (7) state that the postconditions of thread π' ($\mathsf{Q}_\mathsf{s}^{\pi'}$ and $\mathsf{Q}_\mathsf{f}^{\pi'}$) are stable with respect to the actions of all threads. Observe that in the case of (5), despite the additional stipulation $\pi \notin \overline{\pi'}$, the actions of π are irrelevant and do not affect the stability of $\mathsf{P}^{\pi'}$. More concretely, the precondition $\mathsf{P}^{\pi'}$ only holds at the beginning of the program *before* new descendants are spawned (line 8). As such, at these program points $\mathsf{P}^{\pi'}$ is trivially stable with respect to the actions of its (non-existing) descendants.

Spatial representation

$$\mathsf{Pre}(\gamma_0,t,x,\pi) \stackrel{\text{def}}{=} \overline{\pi} * \boxed{\exists\gamma.\, g \Rightarrow \gamma * (\gamma \dot{\leqslant}_t \gamma_0 \wedge \mathsf{Inv}(\gamma,t) \wedge \mathsf{P}^\pi(\gamma,t,x)) * \mathsf{g}(\gamma,x)}_I$$

$$\mathsf{Post}(\gamma_0,t,x,\pi,b) \stackrel{\text{def}}{=} \overline{\pi} * \begin{pmatrix} b \dot{=} 1 * \boxed{\exists\gamma.\, g \Rightarrow \gamma * (\gamma \dot{\leqslant}_t \gamma_0 \wedge \mathsf{Inv}(\gamma,t) \wedge \mathsf{Q}_\mathsf{s}^\pi(\gamma,x)) * \mathsf{t}(\gamma,x,\pi)}_I \\ \vee\ b \dot{=} 0 * \boxed{\exists\gamma.\, g \Rightarrow \gamma * (\gamma \leqslant_t \gamma_0 \wedge \mathsf{Inv}(\gamma,t) \wedge \mathsf{Q}_\mathsf{f}^\pi(\gamma,x)) * \mathsf{g}(\gamma,x)}_I \end{pmatrix}$$

$$\mathsf{g}(\gamma,x) \stackrel{\text{def}}{=} (x{=}0 \wedge \mathsf{emp}) \vee \left(\bigstar_{x \overset{\gamma}{\rightsquigarrow}^* y} \mathsf{node}(\gamma,y) \right)$$

$$\mathsf{t}(\gamma,x,\pi) \stackrel{\text{def}}{=} (x{=}0 \wedge \mathsf{emp}) \vee (\mathsf{M}(\gamma,x,\pi) * \mathsf{t}(\gamma,\gamma^\mathsf{l}(x),\pi.\mathsf{l}) * \mathsf{t}(\gamma,\gamma^\mathsf{r}(x),\pi.\mathsf{r}))$$

$$\mathsf{node}(\gamma,x) \stackrel{\text{def}}{=} \mathsf{M}(\gamma,x,{-}) \vee \mathsf{U}(\gamma,x)$$

$$\mathsf{M}(\gamma,x,m) \stackrel{\text{def}}{=} m \dot{=} \gamma^\mathsf{m}(x) * m \dot{\in} \Pi * \exists l,r.\, \gamma^\mathsf{l}(x) \dot{=} l * \gamma^\mathsf{r}(x) \dot{=} r * x \mapsto 1,l,r * x \Rightarrow m$$

$$\mathsf{U}(\gamma,x) \stackrel{\text{def}}{=} \gamma^\mathsf{m}(x) \dot{=} 0 * \exists l,r.\, \gamma^\mathsf{l}(x) \dot{=} l * \gamma^\mathsf{r}(x) \dot{=} r * x \mapsto 0,l,r * x \Rightarrow 0$$

$$I \stackrel{\text{def}}{=} \{\pi : \mathsf{g}(\gamma,x) \wedge \gamma\, A_\pi\, \gamma' \rightsquigarrow \mathsf{g}(\gamma',x)$$

We use the following lemma when verifying span to fold and unfold the graph predicate g.

Lemma 5. *For all mathematical graphs γ and nodes x:*

$$\mathsf{g}(\gamma,x) \iff (x{=}0 \wedge \mathsf{emp}) \vee (\mathsf{node}(\gamma,x) \circledast \mathsf{g}(\gamma,\gamma^\mathsf{l}(x)) \circledast \mathsf{g}(\gamma,\gamma^\mathsf{r}(x)))$$

Verifying span A proof sketch of span is given in Fig. 10.

Figure 10: Code and proof sketch of the concurrent spanning tree program.