Verifying Graph Algorithms in Separation Logic: A Case for an Algebraic Approach

MARCOS GRANDURY, IMDEA Software Institute, Spain and Universidad Politécnica de Madrid, Spain
ALEKSANDAR NANEVSKI, IMDEA Software Institute, Spain
ALEXANDER GRYZLOV, IMDEA Software Institute, Spain

Verifying graph algorithms has long been considered challenging in separation logic, mainly due to structural sharing between graph subcomponents. We show that these challenges can be effectively addressed by representing graphs as a partial commutative monoid (PCM), and by leveraging structure-preserving functions (PCM morphisms), including higher-order combinators.

PCM morphisms are important because they generalize separation logic's principle of local reasoning. While traditional framing isolates relevant portions of the heap only at the top level of a specification, morphisms enable contextual localization: they distribute over monoid operations to isolate relevant subgraphs, even when nested deeply within a specification.

We demonstrate the morphisms' effectiveness with novel and concise verifications of two canonical graph benchmarks: the Schorr-Waite graph marking algorithm and the union-find data structure.

CCS Concepts: • **Theory of computation → Separation logic**; **Program verification**.

Additional Key Words and Phrases: Graphs, Partial Commutative Monoids, PCM morphisms, Combinators

ACM Reference Format:
Marcos Grandury, Aleksandar Nanevski, and Alexander Gryzlov. 2025. Verifying Graph Algorithms in Separation Logic: A Case for an Algebraic Approach. *Proc. ACM Program. Lang.* 9, ICFP, Article 241 (August 2025), 30 pages. https://doi.org/10.1145/3747510

1 Introduction

The defining property of separation logic [Ishtiaq and O'Hearn 2001; O'Hearn et al. 2001; Reynolds 2002] is that a program's specification tightly circumscribes the heap that the program accesses. Then *framing*, with the associated frame rule of inference, allows extending the specification with a *disjoint* set of pointers, deducing that the program doesn't modify the extension. This makes the verification *local* [Calcagno et al. 2007], in the sense that it can focus on the relevant heap by framing out the unnecessary pointers.

Another characteristic property of separation logic is that data structure's layout in the heap is typically defined in relation to the structure's contents, so that clients can reason about the contents and abstract away from the heap. A common example is the predicate *list* α (i, j) [Reynolds 2002] which holds of a heap iff that heap contains a singly-linked list between pointers i and j, and the list's contents corresponds to the inductive mathematical sequence α. This way, *spatial* (i.e., about state, heaps and pointers) reasoning gives rise to *non-spatial* (i.e., mathematical, state-free, about contents) reasoning for the clients. Fig. 1 illustrates a heap satisfying *list* α (i, j) when $\alpha = [a, b, c, d]$.

Authors' Contact Information: Marcos Grandury, IMDEA Software Institute, Madrid, Spain and Universidad Politécnica de Madrid, Madrid, Spain, marcos.grandury@imdea.org; Aleksandar Nanevski, IMDEA Software Institute, Madrid, Spain, aleks.nanevski@imdea.org; Alexander Gryzlov, IMDEA Software Institute, Madrid, Spain, aliaksandr.hryzlou@imdea.org.

ACM 2475-1421/2025/8-ART241
https://doi.org/10.1145/3747510

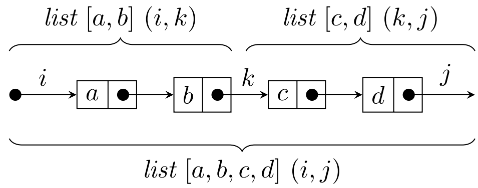

Fig. 1. Spatial predicate *list* α (i, j) describes the layout of a linked list, whose contents is sequence α, between pointers i and j. Dividing the heap is matched by dividing α.

The interaction of the above two properties imparts an important requirement on non-spatial reasoning. Since framing involves decomposition of heaps into disjoint subheaps, locality mandates that the mathematical structure representing contents also needs to support some form of disjoint decomposition. For example, in Fig. 1, the heap divides into two contiguous but disjoint parts, between pointers i and k, and k and j, respectively. Correspondingly, at the non-spatial level, the sequence $[a, b, c, d]$ laid out between the pointers i and j, decomposes into subsequences $[a, b]$ and $[c, d]$, such that *list* $[a, b]$ (i, k) and *list* $[c, d]$ (k, j) hold of disjoint subheaps.

While mathematical sequences (along with sets, mathematical trees, and many other structures) admit useful forms of decomposition, graphs generally don't. Indeed, traversing a list or a tree reduces to traversing disjoint sublists or disjoint subtrees from different children of a common root; thus lists and trees naturally divide into disjoint subcomponents. But in a graph, the subgraphs reachable from the different children of a common node need not be disjoint, preventing an easy division. Even if a graph is divided into subgraphs with disjoint nodes, one must still somehow keep track of the edges crossing the divide, in order to eventually reattach the pieces. Keeping track of the crossing edges is simple in the case of lists, as there's at most one such edge (e.g., k in Fig. 1), but graphs generally don't exhibit such regularity.

The lack of natural decomposition is why graphs have traditionally posed a challenge for separation logic. For example, the Schorr-Waite graph marking algorithm [Schorr and Waite 1967] is a well-known graph benchmark,[1] that has been verified in separation logic early on by Yang [2001a,b], and in many other logics and tools such as Isabelle [Mehta and Nipkow 2003] and Dafny [Leino 2010]. While these proofs share conceptually similar mathematical notions (e.g., reachability in a graph by paths of unmarked nodes), they are formally encoded differently, with Yang's proof exhibiting by far the largest formalization overhead.[2]

Notably, Yang's proof avoids using graphs in non-spatial reasoning by refraining from explicit mathematical graph parameters in spatial predicates. Instead, these predicates are parametrized by decomposable abstractions—such as node sequences encoding traversal paths, marked/unmarked node sets, the graph's spanning tree, etc.—that indirectly approximate the graph's structure. However, these proxies are interdependent: maintaining their mutual consistency and synchronization with the heap-allocated graph forces the proof to perpetually alternate between spatial and non-spatial reasoning, an entanglement that accounts for most of the proof's complexity. Furthermore, the absence of an explicit mathematical graph parameter precludes invoking formal graph-theoretic lemmas, which in turn limits proof reuse and scalability.

[1]The Schorr-Waite algorithm, described in Section 4, marks the nodes reachable from a given root node in a directed graph. Unlike recursive depth-first implementation, it uses an iterative approach that simulates the call stack through graph mutations, temporarily modifying the graph's pointer structure during traversal and restoring it upon completion.

[2]Yang's proof is described in some 40 pages of dense mathematical text. The proof in Isabelle is mechanized in about 400 lines, and the proof in Dafny is automated, aside from some 70 lines of annotations. Of course, Yang's proof is the only one to use framing, which allowed restricting the focus to connected graphs without sacrificing generality.

To reconcile spatial and non-spatial reasoning, we first require a mathematical representation of graphs that inherently supports decomposition. This motivates our generalization to *partial graphs*, which are directed graphs that admit *dangling edges*, i.e., edges whose source node belongs to the graph, but whose sink node doesn't. Partial graphs decompose into subgraphs with disjoint nodes, where an edge crossing the divide becomes dangling in the subgraph containing the edge's source. We formalize this decomposition by casting partial graphs as *partial commutative monoid (PCM)*. A PCM is a structure $(U, \bullet, e)$ where $\bullet$, pronounced "join", is a partial commutative and associative binary operation over U, with unit e, which captures how graphs combine. If $\gamma = \gamma_1 \bullet \gamma_2$, we say that γ *decomposes* or *splits* into γ_1 and γ_2, or that γ_1 and γ_2 *join* into γ.

PCMs are central to separation logic, e.g., for defining heaps [Calcagno et al. 2007], permissions [Bornat et al. 2005], histories [Sergey et al. 2015b], and meta theory [Dinsdale-Young et al. 2013]. Modern separation logics further support abstract states described by arbitrary PCMs [Jung et al. 2015; Krishna et al. 2020; Nanevski et al. 2019]. However PCMs *alone* are insufficient for graph verification: while they enable spatial reasoning over diverse notions of state, effective graph verification requires a mechanism to localize *non-spatial* graph proofs.

We address this gap with the three novel contributions of this paper: (1) leveraging PCM morphisms in graph reasoning, (2) utilizing higher-order morphisms (i.e., combinators), leading to (3) new proofs of Schorr-Waite and of another graph benchmark—the union-find data structure [Charguéraud and Pottier 2019; Krishnaswami 2011; Wang et al. 2019].

Leveraging morphisms in graph reasoning. Given PCMs A and B, the function $f : A \longrightarrow B$ is a PCM morphism if the following equations hold.

$$f\ e_A \ = \ e_B \tag{1}$$

$$f\ (\gamma_1 \bullet_A \gamma_2) \ = \ f\ \gamma_1 \bullet_B f\ \gamma_2 \tag{2}$$

Focusing on equation (2), it says that f distributes over the join operation of the domain PCM and determines the following two ways in which morphisms apply to verification.

First, morphisms mediate between PCMs, as the join operation of A is mapped to the join operation of B. We will use morphisms between various PCMs in this paper, but an important example is to morph the PCM of graphs to the PCM of propositions of separation logic (equivalently, to the PCM of sets of heaps). Connecting graphs to sets of heaps allows a suitable heap modification to be abstracted as a modification of the graph, thereby elevating low-level separation logic reasoning about pointers into higher-level reasoning about graphs. For example, if a node in a graph is represented as a heap storing the node's adjacency list, mutating the pointers in this heap corresponds at the graph level to modifying the node's edges.

Second, continuing with morphisms over graphs specifically, the value of f over a graph decomposed into components (γ_1 and γ_2 in equation (2)) can be computed by applying f to γ_1 and γ_2 *independently* and joining the results. During verification, the left-to-right direction of (2) localizes reasoning to the modified subgraph (say, γ_1), while the right-to-left direction automatically propagates and reattaches γ_2. This allows morphisms to serve an analogous localizing role to framing in separation logic, but with two distinctions. The obvious one is that framing decomposes heaps, whereas morphisms decompose mathematical graphs (spatial vs. non-spatial reasoning). More importantly, framing operates only at the top level of specifications, whereas morphisms localize deeply inside a context: equation (2) can rewrite within arbitrary context $I(-)$ to transform $I(f(\gamma_1 \bullet \gamma_2))$ into $I(f\gamma_1 \bullet f\gamma_2)$ and allow $f\gamma_1$ and $f\gamma_2$ to be manipulated independently inside $I(-)$. The latter is the essential feature that differentiates non-spatial from spatial reasoning (where top-level framing suffices), and we shall use it to enable and streamline graph proofs.

As an illustration of morphisms, consider *filtering* a graph to obtain a subgraph containing only the nodes with a specific property. Filtering differs from taking a subgraph in standard graph theory, in that the filtered subgraph retains—as dangling—the edges into the part avoided by the filter. This enables reattaching the two parts later on, after either has been processed. Clearly, to filter a composite graph it suffices to filter the components and join the results, making filtering a PCM morphism on partial graphs.

While not all useful graph abstractions are morphisms, those that aren't can still usefully interact with morphisms. The canonical example of such *global* notions is *reachability* between nodes in a graph [Krishna et al. 2020]. In verification, reachability often needs to be restricted to paths traversing only specific (e.g., unmarked) nodes. While one could implement this restriction by parametrizing reachability with admissible node sets (e.g., as done by Leino [2010]), a simpler solution in the presence of morphisms is to evaluate the global reachability predicate on dynamically filtered subgraphs—effectively composing reachability with filtering. This compositional approach scales naturally, enabling us to express complex specifications (like Schorr-Waite's) by combining elementary graph operations from a minimal core vocabulary.

Higher-order combinators. In the presence of morphisms, the specifications can utilize still higher levels of abstraction. Our second contribution notes that useful graph transformations are encodable as instances of a *higher-order combinator* (like map over sequences in functional programming), itself a PCM morphism on partial graphs. Morphisms thus relate different levels of the abstraction stack that has heaps at the bottom, and localize the reasoning across all levels.

New proofs of Schorr-Waite and union-find. The main challenge in these proofs is establishing the invariance of complex global properties, which we address precisely using contextual localization, morphisms and combinators. Based on these abstractions, we also develop a generic theory of partial graphs, which reduces example-specific reasoning and yields conceptually simple and concise proofs. The entire development (graph theory, Schorr-Waite, union-find) is mechanized in Hoare Type Theory [HTT 2010; Nanevski et al. 2006, 2008, 2010], a Coq library for separation logic reasoning via types, with the component sizes summarized below [Grandury et al. 2025a].

	lines of proof	lines of specification (definitions, annotations, notation, code)
Graph-theory library	2070	1738
Schorr-Waite	110	111
Union-find	49	46

Finally, we note that working with morphisms imparts a distinct algebraic character to specifications and proofs in this paper. While our reasoning spans both spatial and non-spatial domains, the bulk of the effort is on the non-spatial side, where decomposition is governed by the PCM operation $\bullet$ rather than the separation logic's characteristic spatial connective $*$. This shift manifests in specifications: assertions use $*$ sparingly as decomposition occurs primarily via $\bullet$ at the abstract graph level. It's this prioritization of the PCM structure and morphisms over traditional spatial decomposition that justifies the term "algebraic" in the paper's title.

We adopt the classical separation logic formulation from O'Hearn et al. [2001], Yang's dissertation [2001b] and Reynolds [2002], whose inference rules (e.g., frame rule, consequence, etc.) are standard and thus elided here. However, to enable the interleaving of heap-level assertions with graph-theoretic transformations, we depart somewhat from that work by allowing the assertions to embed unrestricted mathematical formulas, including direct applications of morphisms.

All appendix references in the paper point to the extended version [Grandury et al. 2025b].

1. $\{\exists \alpha\ \beta.\ \mathit{list}\ \alpha\ (i, j) * \mathit{list}\ \beta\ (j, \text{null}) \wedge n = \#\alpha \wedge \alpha_0 = \alpha \bullet \beta \wedge j \neq \text{null}\}$
2. $\{\exists \alpha\ b\ \beta'.\ \mathit{list}\ \alpha\ (i, j) * \mathit{list}\ (b \bullet \beta')\ (j, \text{null}) \wedge n = \#\alpha \wedge \alpha_0 = \alpha \bullet (b \bullet \beta')\}$
3. $\{\exists \alpha\ b\ \beta'\ k.\ \mathit{list}\ \alpha\ (i, j) * j \Mapsto b, k * \mathit{list}\ \beta'\ (k, \text{null}) \wedge n = \#\alpha \wedge \alpha_0 = \alpha \bullet (b \bullet \beta')\}$
4. $j := j.\text{next};$
5. $\{\exists \alpha\ b\ \beta'\ j'.\ \mathit{list}\ \alpha\ (i, j') * j' \Mapsto b, j * \mathit{list}\ \beta'\ (j, \text{null}) \wedge n = \#\alpha \wedge \alpha_0 = \alpha \bullet (b \bullet \beta')\}$
6. $\{\exists \alpha\ b\ \beta'.\ \mathit{list}\ (\alpha \bullet b)\ (i, j) * \mathit{list}\ \beta'\ (j, \text{null}) \wedge n = \#\alpha \wedge \alpha_0 = (\alpha \bullet b) \bullet \beta'\}$
7. $n := n + 1;$
8. $\{\exists \alpha\ b\ \beta'.\ \mathit{list}\ (\alpha \bullet b)\ (i, j) * \mathit{list}\ \beta'\ (j, \text{null}) \wedge n = \#\alpha + 1 \wedge \alpha_0 = (\alpha \bullet b) \bullet \beta'\}$
9. $\{\exists \alpha\ b\ \beta'.\ \mathit{list}\ (\alpha \bullet b)\ (i, j) * \mathit{list}\ \beta'\ (j, \text{null}) \wedge n = \#(\alpha \bullet b) \wedge \alpha_0 = (\alpha \bullet b) \bullet \beta'\}$
10. $\{\exists \alpha'\ \beta'.\ \mathit{list}\ \alpha'\ (i, j) * \mathit{list}\ \beta'\ (j, \text{null}) \wedge n = \#\alpha' \wedge \alpha_0 = \alpha' \bullet \beta'\}$

Fig. 2. Proof outline that the loop body of the length-computing program L preserves the loop invariant I.

2 Background

To pinpoint the patterns of separation logic that inform our approach to graphs, we consider the following program for computing the length of a singly-linked list headed at the pointer i.

$$L\ \widehat{=}\ n := 0; j := i; \textbf{while}\ j \neq \text{null}\ \textbf{do}\ j := j.\text{next}; n := n + 1\ \textbf{end while}$$

To specify L, one must first describe how a singly-linked list is laid out in the heap. For that, the proposition $\mathit{list}\ \alpha\ (i, j)$ from Section 1 is defined below to hold of a heap that stores a linked list segment between pointers i and j, whose contents is the mathematical sequence α. Following Reynolds [2002], we overload $\bullet$ to denote attaching an element to a head or a tail of a sequence, and concatenating two sequences.

$$\begin{aligned} \mathit{list}\ []\ (i, j) &\ \widehat{=}\ \mathit{emp} \wedge i = j \\ \mathit{list}\ (a \bullet \alpha)\ (i, j) &\ \widehat{=}\ \exists k.\ i \Mapsto a, k * \mathit{list}\ \alpha\ (k, j) \end{aligned}$$

The definition is inductive in α and says: (1) The empty sequence $[]$ is stored in a heap between pointers i and j iff the heap is empty and $i = j$; (2) The sequence $a \bullet \alpha$ is stored in the heap between pointers i and j iff i points to a list node storing a in the value field, and some pointer k in the next field, so that α is then stored between k and j in a heap segment *disjoint* from i.

Here *emp*, $\Mapsto$ and $*$ are the spatial propositional connectives of separation logic: *emp* holds of a heap iff the heap is empty; $i \Mapsto v_0, \ldots, v_n$ holds of a heap that contains only the pointers $i, \ldots, i + n$ storing the values $v_0, \ldots, v_n$, respectively; the separating conjunction $P * Q$ holds of a heap that can be divided into two disjoint subheaps of which P and Q hold respectively. Separation logic propositions form a PCM with $*$ as the commutative/associative operation, and *emp* its unit.

The following Hoare triple then applies the *sequence length* function $\#(-)$ to say that, upon L's termination, the contents α_0 of the initial null-terminated list is unchanged, but its length $\#\alpha_0$ is deposited into the variable n.

$$\{\mathit{list}\ \alpha_0\ (i, \text{null})\}\ L\ \{\mathit{list}\ \alpha_0\ (i, \text{null}) \wedge n = \#\alpha_0\}$$

We next outline the part of the proof for this Hoare triple that examines the loop body in L, highlighting three key high-level aspects that we later adapt to graphs.

Dangling pointers. The first aspect is that separation logic inherently relies on *dangling pointers* to capture computations' intermediate states. For example, the loop invariant for L is

$$I \mathrel{\widehat{=}} \exists \alpha\, \beta.\, \mathit{list}\ \alpha\ (i, j) * \mathit{list}\ \beta\ (j, \mathrm{null}) \wedge n = \#\alpha \wedge \alpha_0 = \alpha \bullet \beta$$

stating that the original sequence α_0 divides into α (processed subsequence) and β (remaining subsequence), with n tracking α's length (progress computed so far). Importantly, pointer j connects α's tail to β's head, making it dangling for α since it references memory outside of α's heap.

Spatial distributivity. The second aspect is that the *list* predicate distributes over $\bullet$, in the sense of the following equivalence, already illustrated in Fig. 1.

$$\mathit{list}\ (\alpha \bullet \beta)\ (i, j) \iff \exists k.\, \mathit{list}\ \alpha\ (i, k) * \mathit{list}\ \beta\ (k, j) \tag{3}$$

Equation (3) is clearly related to the distributivity of morphisms (2), even though sequences don't form a PCM as concatenation isn't commutative. Nonetheless, the equation underscores that distributivity, in one form or another, is a crucial notion in separation logic. It's typical use in separation logic is to transfer ownership of pointers between heaps; in the case of L, to redistribute *list* so that the currently counted element, pointed to by j, is moved in assertions from β to α.

To illustrate, we review the proof in Fig 2 that I is the loop invariant for L. Line 1 conjoins I with the loop condition $j \neq \mathrm{null}$, to indicate that the execution is within the loop. Line 2 derives that β is non-empty, hence of the form $b \bullet \beta'$, as otherwise j would have been null by the definition of *list*. Line 3 shows the first use of distributivity to *detach* b from the head of β. Line 4 mutates j into j.next, which is reflected in line 5, where a fresh variable j' names the value of j before the mutation. This line derives by the standard inference rules for pointer mutation and framing [Reynolds 2002]. Line 6 shows the second use of distributivity to *attach* b to the tail of α; it also reassociates the concatenations in α_0 correspondingly. Line 7 increments n, which is reflected in lines 8 and 9. Finally, line 10 re-establishes I for the updated values of j and n.

Non-spatial distributivity. The third aspect is that the distribution over $\bullet$ is important for the non-spatial parts of the proof as well. In particular, transitioning from line 8 to line 9 requires that the length function distributes over $\bullet$; specifically, that $\#(\alpha \bullet b) = \#\alpha + 1$, or more generally, that

$$\#(\alpha \bullet \beta) = \#\alpha + \#\beta$$

to associate the new value of n to the length of the new processed sequence α'. The transformation occurs in the context $n = (-)$ in lines 8–9, illustrating a simple case of contextual localization.

While standard treatments of separation logic rely on non-spatial distributivity only implicitly, graph verification requires that it be made explicit and central. This is because graph specifications commonly compose morphisms, in turn making contextual localization essential for effective proofs. By bringing these algebraic foundations—distributivity and morphisms—to the foreground, our approach allows them to be systematically leveraged in verification.

3 Partial Graphs

Dangling edges. The example in Section 2 shows how dangling *pointers* link disjoint list segments to specify intermediate computation states. Similarly, specifying graph algorithms requires dangling *edges* to connect disjoint parts of a graph. However, unlike lists, which have a single dangling pointer, graphs generally have multiple dangling edges bridging the divide (Fig 3). To parallel Section 2, where the dangling pointer j was treated as a parameter of *list*, one might consider grouping the dangling edges into a set that parametrizes the predicate *graph*, the graph analogue to *list*. Bornat et al. [2004] explored this approach, but it resulted in an unsatisfactory definition, as the resulting predicate relied on a program-specific traversal order in addition to the graph itself.

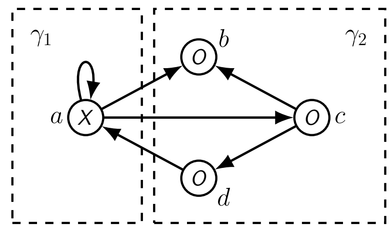

Fig. 3. Graph decomposition. Node a has mark X, and nodes b, c, d have mark O. Graph γ_1 has dangling edges from a to b and c; graph γ_2 has a dangling edge from d to a.

We instead *embed* dangling edges directly into the graph representation, yielding the *partial graphs* informally introduced in Section 1. Specifically, we choose a graph representation where each node is associated with its adjacency list, but we allow for the possibility that a node in the adjacency list *need not be present in the graph itself.* More formally, a partial graph (or simply graph) of type T is a *partial finite map* on nodes (isomorphic to natural numbers), that, if defined on a node x, maps x to a pair consisting of a value of type T (x's contents, value, or mark) and the sequence of nodes adjacent to x (x's immediate successors, adjacency list/sequence, children), with the proviso that the map is undefined on the node null (i.e. 0).

$$\textit{partial-graph}\ T \ \widehat{=}\ \textit{node} \rightharpoonup_{\textit{fin}} T \times \textit{seq node}$$

The *domain* of a map is the finite subset of the type *node* on which the map is defined. A graph contains an edge from x to y if y appears among the children of x in the map. An edge from x to y is dangling if y isn't in the map's domain. We write *binary-graph* T for the subtype of *partial-graph* T where the adjacency list of each node has exactly two elements (left/right child), with the element set to null if the corresponding child doesn't exist. Notation $x \mapsto (v, \alpha)$ denotes the *singleton graph*, comprising just the node x, with mark v and adjacency list α. The notation simplifies to $x \mapsto \alpha$ when T is the unit type, as then the mark v isn't important.

For example, the graph γ_1 in Fig. 3 consists of a single node a with contents X, and adjacency list $[a, b, c]$, capturing that a has edges to itself, and to nodes b and c. The latter edges are dangling, as b and c are outside of γ_1. In our notation, $\gamma_1 = a \mapsto (X, [a, b, c])$.

Spatial distributivity. The definition also directly leads to a notion of graph (de)composition. To see this, notice that finite maps admit the operation of disjoint union which combines the maps γ_1 and γ_2, but only if γ_1 and γ_2 have disjoint domains.

$$\gamma_1 \bullet \gamma_2 \ \widehat{=} \begin{cases} \gamma_1 \cup \gamma_2 & \text{if } \textit{dom}\ \gamma_1 \cap \textit{dom}\ \gamma_2 = \emptyset \\ \text{undefined} & \text{otherwise} \end{cases}$$

The operation is denoted $\bullet$ to draw the analogy with consing/concatenating in the case of sequences, and also because $\bullet$ is partial, commutative and associative, meaning that graphs with $\bullet$ form a PCM. The unit e is the empty graph, i.e. the everywhere-undefined map.

For example, the full graph in Fig. 3 is a composition $\gamma_1 \bullet \gamma_2$, where the subgraph γ_1 has already been described, and $\gamma_2 = b \mapsto (O, []) \bullet c \mapsto (O, [b, d]) \bullet d \mapsto (O, [a])$. The representation captures, among other properties, that γ_2 has a node d with a dangling edge to a. However, the composite graph $\gamma_1 \bullet \gamma_2$ has no edges dangling.

Similarly to *list*, we can now define the predicate *graph* that defines how a graph is laid out in the heap. This can be achieved in several different ways, optimizing for the structure of the graphs of interest. We will conflate nodes with pointers and, for general partial graphs, lay out the adjacency list of each node as a linked list in the heap.

$$\begin{aligned} \mathit{graph}\, e \;&\widehat{=}\; \mathit{emp} \\ \mathit{graph}\,(x \mapsto (v, \alpha) \bullet \gamma) \;&\widehat{=}\; \exists i.\, x \Mapsto v, i * \mathit{list}\ \alpha\ i\ \mathrm{null} * \mathit{graph}\,\gamma \end{aligned}$$

For binary graphs, however, a simpler representation avoids linking altogether. Because adjacency lists have exactly two children, a node x can be represented as a heap with three cells: the pointer x storing the mark v, with the *subsequent* pointers $x + 1$ and $x + 2$ pointing to x's children.

$$\begin{aligned} \mathit{graph}\, e \;&\widehat{=}\; \mathit{emp} \\ \mathit{graph}\,(x \mapsto (v, [l, r]) \bullet \gamma) \;&\widehat{=}\; x \Mapsto v, l, r * \mathit{graph}\,\gamma \end{aligned}$$

The following equation (4) characterizes *both* definitions of *graph* as PCM morphisms from partial graphs to separation logic propositions. This lifts heap framing and separation logic reasoning to graphs, much like *list* did for sequences with equation (3).

$$\mathit{graph}\,(\gamma_1 \bullet \gamma_2) \iff \mathit{graph}\,\gamma_1 * \mathit{graph}\,\gamma_2 \tag{4}$$

To streamline the presentation, in the remainder of the paper we focus on binary graphs. The simpler definition of *graph* will let us concentrate on non-spatial reasoning, where partial graphs interact with PCMs beyond separation logic propositions, and where contextual localization requires the use of morphisms. We consider the basics of this interaction next.

Non-spatial distributivity. We first introduce some common notation and write: *nodes* γ instead of *dom* γ for the set of nodes on which γ is defined as a finite map, to emphasize that γ is a graph; $\mathit{nodes}_0\ \gamma$ for the disjoint union $\{\mathrm{null}\} \cup \mathit{nodes}\ \gamma$; and $\gamma_{\mathrm{val}}\, x$ and $\gamma_{\mathrm{adj}}\, x$, respectively, for the contents and the adjacency list of the node x, so that $\gamma\, x = (\gamma_{\mathrm{val}}\, x, \gamma_{\mathrm{adj}}\, x)$.

Given the graph γ and node x, we define $\gamma \backslash x$ to be the graph that removes x and its outgoing edges from γ, but keep the edges sinking into x as newly dangling. We can characterize $\gamma \backslash x$ as a function as follows: $\gamma \backslash x$ agrees with γ on all inputs except possibly x, on which $\gamma \backslash x$ is undefined.

We can now state the following two important equalities that expand (alt.: unfold) the graph γ.

$$\gamma = (x \mapsto \gamma\, x) \bullet (\gamma \backslash x) \qquad \text{if } x \in \mathit{nodes}\,\gamma \tag{5}$$

$$\gamma = \bigodot_{x \in \mathit{nodes}\,\gamma} x \mapsto \gamma\, x \tag{6}$$

Equality (5) expands the graph γ around a specific node x, so that x's contents and adjacency list can be considered separately from the rest of the graph. This is analogous to how a pointer was separated from *list* in Section 2, so that it could be transferred from one sequence to another. The equality (6) iterates the expansion to characterize the graph in terms of the nodes.

We also require the following functions and combinators over graphs.

$$\gamma/_S \;\widehat{=}\; \bigodot_{x \in S \cap \mathit{nodes}\,\gamma} x \mapsto \gamma\, x \qquad (\mathit{filter}) \tag{7}$$

$$|\gamma| \;\widehat{=}\; \bigodot_{x \in \mathit{nodes}\,\gamma} x \mapsto \gamma_{\mathrm{adj}}\, x \qquad (\mathit{erasure}) \tag{8}$$

$$\mathit{map}\ f\ \gamma \;\widehat{=}\; \bigodot_{x \in \mathit{nodes}\,\gamma} x \mapsto f\ x\ (\gamma\, x) \qquad (\mathit{map}) \tag{9}$$

$$\gamma/_{v_1,\ldots,v_n} \;\widehat{=}\; \gamma/_{\gamma_{\mathrm{val}}^{-1}\{v_1,\ldots,v_m\}} \qquad (\mathit{filter\ by\ contents}) \tag{10}$$

Filtering takes a subset of the nodes of γ that are also in the set S, without modifying the nodes' contents or adjacency lists. Erasure replaces the contents of each node with the singleton element of unit type, thus eliding the contents from the notation. In particular $|-|$: *partial-graph T* $\longrightarrow$ *partial-graph unit*. Map modifies the contents and the adjacency list of each node according to the mapped function. If f : *node* $\longrightarrow T_1 \times$ *seq node* $\longrightarrow T_2 \times$ *seq node* then *map f* : *partial-graph* $T_1 \longrightarrow$ *partial-graph* T_2.[3] Filtering by contents selects the nodes whose contents is one of $v_1, \ldots, v_n$, by (plain) filtering over $\gamma_{\mathrm{val}}^{-1}\{v_1, \ldots, v_n\}$. The latter is the inverse image of γ_{val}; thus, the set of nodes that γ_{val} maps into $\{v_1, \ldots, v_n\}$.

Sinks of a graph is the set of nodes in the range of γ (i.e., nodes that possess an incoming edge).

$$\mathit{sinks}\ \gamma \ \widehat{=} \bigcup_{\mathit{nodes}\ \gamma} \gamma_{\mathrm{adj}} \tag{11}$$

We then say that *closed* γ holds iff γ contains no dangling edges. A plain (i.e., standard, non-partial) graph is a partial graph that is closed.

$$\mathit{closed}\ \gamma \ \widehat{=}\ \mathit{sinks}\ \gamma \subseteq \mathit{nodes}_0\ \gamma \tag{12}$$

The final graph primitive in our vocabulary is *reach* $\gamma\ x$, which computes the set of nodes in γ reachable from the node x. The definition is recursive, unioning x with the nodes reachable from every child of x via a path that avoids x. The definition is well-founded because the (finite) set of graph's nodes decreases with each recursive call, and thus eventually becomes empty.

$$\mathit{reach}\ \gamma\ x \ \widehat{=} \begin{cases} \{x\} \cup \bigcup_{\gamma_{\mathrm{adj}}\ x} \mathit{reach}\ (\gamma \backslash x) & \text{if } x \in \mathit{nodes}\ \gamma \\ \emptyset & \text{otherwise} \end{cases} \tag{13}$$

Fig. 4 illustrates the definitions on the graph $\gamma_1 \bullet \gamma_2$ from Fig. 3. Erasure, *map*, both filters, *nodes* and *sinks* distribute over $\bullet$ (and are actually morphisms), while *closed* and *reach* don't.

Lemma 3.1 (Morphisms). *Functions nodes, $(-)/_S$, $(-)/_{v_i}$, $|-|$, map f, and sinks are morphisms from the PCM of graphs, to an appropriate target PCM (sets with disjoint union for nodes, graphs for $(-)/_S$, $(-)/_{v_i}$, $|-|$, map f, and sets with plain union for sinks).*

1. *nodes* $(\gamma_1 \bullet \gamma_2) =$ *nodes* $\gamma_1 \uplus$ *nodes* γ_2 *and nodes* $e = \emptyset$
2. $(\gamma_1 \bullet \gamma_2)/_S = \gamma_1/_S \bullet \gamma_2/_S$ *and* $e/_S = e$
3. $(\gamma_1 \bullet \gamma_2)/_{v_1,\ldots,v_n} = \gamma_1/_{v_1,\ldots,v_n} \bullet \gamma_2/_{v_1,\ldots,v_n}$ *and* $e/_{v_1,\ldots,v_n} = e$
4. $|\gamma_1 \bullet \gamma_2| = |\gamma_1| \bullet |\gamma_2|$ *and* $|e| = e$
5. *map* $f\ (\gamma_1 \bullet \gamma_2) =$ *map* $f\ \gamma_1 \bullet$ *map* $f\ \gamma_2$ *and map* $f\ e = e$
6. *sinks* $(\gamma_1 \bullet \gamma_2) =$ *sinks* $\gamma_1 \cup$ *sinks* γ_2 *and sinks* $e = \emptyset$

Lemma 3.2 (Filtering).

1. $\gamma/_{S_1 \uplus S_2} = \gamma/_{S_1} \bullet \gamma/_{S_2}$ *and* $\gamma/_{\emptyset} = e$
2. $\gamma/_{v_1,\ldots,v_m,w_1,\ldots,w_n} = \gamma/_{v_1,\ldots,v_m} \bullet \gamma/_{w_1,\ldots,w_m}$, *for disjoint* $\{v_i\}$, $\{w_j\}$
3. $\gamma/_{S_1 \cap S_2} = \gamma/_{S_1}/_{S_2}$
4. $\gamma/_{v_1,\ldots,v_m}/_{w_1,\ldots,w_n} = e$, *for disjoint* $\{v_i\}$, $\{w_j\}$

Lemma 3.3 (Mapping). *map* $f_1\ \gamma =$ *map* $f_2\ \gamma$ *iff* $\forall x \in$ *nodes* $\gamma.\ f_1\ x\ (\gamma\ x) = f_2\ x\ (\gamma\ x)$.

Lemma 3.4 (Reachability).

1. *reach* $\gamma\ x =$ *reach* $|\gamma|\ x$
2. *if* $y \notin$ *reach* $\gamma\ x$, *then reach* $\gamma\ x =$ *reach* $(\gamma \backslash y)\ x$

[3] When mapping over binary graphs, we allow f : *node* $\rightarrow T_1 \times$ (*node* $\times$ *node*) $\rightarrow T_2 \times$ (*node* $\times$ *node*), as adjacency lists have exactly two elements. When $T_2 =$ *unit*, we elide T_2 and allow f : *node* $\rightarrow T_1 \times$ (*node* $\times$ *node*) $\rightarrow$ *node* $\times$ *node*.

	γ_1	γ_2	$\gamma_1 \bullet \gamma_2$
$(-)$	$a \mapsto (X, [a, b, c])$	$b \mapsto (O, [])$ $\bullet\ c \mapsto (O, [b, d])$ $\bullet\ d \mapsto (O, [a])$	$a \mapsto (X, [a, b, c])$ $\bullet\ b \mapsto (O, [])$ $\bullet\ c \mapsto (O, [b, d])$ $\bullet\ d \mapsto (O, [a])$
nodes $(-)$	$\{a\}$	$\{b, c, d\}$	$\{a, b, c, d\}$
$(-)/_{\{b\}}$	e	$b \mapsto (O, [])$	$b \mapsto (O, [])$
$(-)/_X$	$a \mapsto (X, [a, b, c])$	e	$a \mapsto (X, [a, b, c])$
$\lvert - \rvert$	$a \mapsto [a, b, c]$	$b \mapsto []$ $\bullet\ c \mapsto [b, d]$ $\bullet\ d \mapsto [a]$	$a \mapsto [a, b, c]$ $\bullet\ b \mapsto []$ $\bullet\ c \mapsto [b, d]$ $\bullet\ d \mapsto [a]$
map $f\ (-)$	$a \mapsto (O, [b, c])$	$b \mapsto (X, [])$ $\bullet\ c \mapsto (X, [d])$ $\bullet\ d \mapsto (X, [])$	$a \mapsto (O, [b, c])$ $\bullet\ b \mapsto (X, [])$ $\bullet\ c \mapsto (X, [d])$ $\bullet\ d \mapsto (X, [])$
sinks $(-)$	$\{a, b, c\}$	$\{a, b, d\}$	$\{a, b, c, d\}$
closed $(-)$	*false*	*false*	*true*
reach $(-)\ a$	$\{a\}$	$\emptyset$	$\{a, b, c, d\}$

Fig. 4. Action of graph abstractions on the graph from Fig. 3. The morphisms are illustrated at the top, and the non-morphic (aka. global) properties at the bottom. In the top part, the mapped function f exchanges O and X and takes the tail of the adjacency list of each node: $f\ x\ (v, \alpha)\ \widehat{=}\ (\overline{v}, \mathit{tail}\ \alpha)$, where $\overline{O} = X$ and $\overline{X} = O$.

(3) *if $y \in \mathit{reach}\ \gamma\ x$, then $\mathit{reach}\ \gamma\ x = \mathit{reach}\ (\gamma \backslash y)\ x \cup \mathit{reach}\ \gamma\ y$*

Lemma 3.5 (Closure).

(1) *If closed γ then closed $(\gamma/_{\mathit{reach}\,\gamma\,x})$, for every x.*

(2) *If closed $(\gamma_1 \bullet \gamma_2)$, and $x \in$ nodes $(\gamma_1 \bullet \gamma_2)$, and nodes $\gamma_1 = \mathit{reach}\ (\gamma_1 \bullet \gamma_2)\ x$ (i.e., γ_1 is the subgraph reachable from x), then closed γ_1, and $x \in$ nodes γ_1, and nodes $\gamma_1 = \mathit{reach}\ \gamma_1\ x$.*

We elide the proofs here (they are in our Coq graph library), and just comment on the intuition behind each. Lemmas 3.1 and 3.2 hold because combinators iterate a node-local transformation over a set of nodes. Lemma 3.3 holds because *map* applies the argument function only to nodes in the graph. Lemma 3.4 (1) holds because reachability isn't concerned with the contents of the nodes. Lemma 3.4 (2) holds because a node y that isn't reachable from x doesn't influence the reachability relation, and can thus be removed from the graph. Lemma 3.4 (3) holds because, given y that's reachable from x, another node z is reachable from x iff it's reachable from y, or is otherwise reachable from x by a path avoiding y. Lemma 3.5 (1) restates in the notation of partial graphs the well-known property that in a standard non-partial graph, the nodes reachable from some node x form a connected subgraph. Lemma 3.5 (2) is a simple consequence of Lemma 3.5 (1).

We close by illustrating how contextual localization helps prove that a global property (here, *closed*) is preserved under graph modifications—a pattern used extensively in the Schorr-Waite and union-find verifications (Sections 4 and 7). The idea is captured in the following lemma and proof.

Lemma 3.6. *Let γ be a binary graph such that closed γ, and $x \in$ nodes γ and $y \in \mathit{nodes}_0\ \gamma$. The graph γ' obtained by modifying x's child (left or right) to y, also satisfies closed γ'.*

Proof. Without loss of generality, we assume that it's the left child of x that's modified. In other words, we take $\gamma = x \mapsto (v, [x_l, x_r]) \bullet \gamma \backslash x$ and $\gamma' = x \mapsto (v, [y, x_r]) \bullet \gamma \backslash x$, for some v, x_l and x_r.

Having assumed *closed* γ, we need to prove *closed* γ'; that is *sinks* $\gamma' \subseteq$ *nodes*$_0$ γ'. While *closed* itself is a global property, the main components of its definition, *sinks* and *nodes*, are morphisms. The proof rewrites within the context $- \subseteq -$ to distribute the morphisms as follows.

$$
\begin{array}{rcll}
\mathit{sinks}\ \gamma' & = & \mathit{sinks}\ (x \mapsto (v, [y, x_r]) \bullet \gamma \backslash x) = & \\
 & = & \mathit{sinks}\ (x \mapsto (v, [y, x_r]) \cup \mathit{sinks}\ \gamma \backslash x & \text{Distributivity of } \mathit{sinks} \\
 & = & \{y, x_r\} \cup \mathit{sinks}\ \gamma \backslash x & \text{Definition of } \mathit{sinks} \\
 & \subseteq & \{y\} \cup \{x_l, x_r\} \cup \mathit{sinks}\ \gamma \backslash x & \\
 & = & \{y\} \cup \mathit{sinks}\ (x \mapsto (v, [x_l, x_r]) \cup \mathit{sinks}\ \gamma \backslash x & \text{Definition of } \mathit{sinks} \\
 & = & \{y\} \cup \mathit{sinks}\ (x \mapsto (v, [x_l, x_r]) \bullet \gamma \backslash x) & \text{Distributivity of } \mathit{sinks} \\
 & = & \{y\} \cup \mathit{sinks}\ \gamma & \\
 & \subseteq & \mathit{nodes}_0\ \gamma & \text{By assumptions } y \in \mathit{nodes}_0\ \gamma \\
 & & & \text{and } \mathit{closed}\ \gamma \text{ (i.e.} \\
 & & & \mathit{sinks}\ \gamma \subseteq \mathit{nodes}_0\ \gamma) \\
 & = & \{\text{null}\} \uplus \mathit{nodes}\ (x \mapsto (v, [x_l, x_r]) \bullet \gamma \backslash x) & \\
 & = & \{\text{null}\} \uplus \mathit{nodes}\ (x \mapsto (v, [x_l, x_r])) \uplus \mathit{nodes}\ \gamma \backslash x & \text{Distributivity of } \mathit{nodes} \\
 & = & \{\text{null}\} \uplus \{x\} \uplus \mathit{nodes}\ \gamma \backslash x & \text{Definition of } \mathit{nodes} \\
 & = & \{\text{null}\} \uplus \mathit{nodes}\ (x \mapsto (v, [y, x_r]) \uplus \mathit{nodes}\ \gamma \backslash x & \text{Definition of } \mathit{nodes} \\
 & = & \mathit{nodes}_0\ (x \mapsto (v, [y, x_r]) \bullet \gamma \backslash x) & \text{Distributivity of } \mathit{nodes} \\
 & = & \mathit{nodes}_0\ \gamma' &
\end{array}
$$

By distributing the morphisms, the proof separates the reasoning about the node x from that about the subgraph $\gamma \backslash x$. While it manipulates the values of *sinks* and *nodes* at x to relate *sinks* γ' and *nodes*$_0$ γ' to *sinks* γ and *nodes*$_0$ γ, respectively, it also exploits the fact that γ and γ' share the same subgraph $\gamma \backslash x$. By isolating the treatment of x from the rest of the graph, the proof enacts a style of reasoning that's fundamentally local, albeit distinct from the typical notion of locality of separation logic that's achieved by framing. This alternative, contextual, form of locality arises not from eliding the rest of the structure, but from explicitly factoring it through morphisms. □

4 Schorr-Waite Algorithm

The goal of a graph-marking algorithm is to traverse a graph starting from some root node r and mark the reachable nodes. An obvious way to implement this functionality is as a recursive function that traverses the graph in a depth-first, left-to-right manner. However, as graph marking is typically employed in garbage collection—when space is sparse—recursive implementation isn't optimal, as it uses up space on the stack to keep track of the recursive calls and execute backtracking. The idea of Schorr-Waite's algorithm is that the information about backtracking can be maintained within the graph itself, while the graph is traversed *iteratively* in a loop.

We consider the variant of Schorr-Waite that operates over binary graphs. We also record the node's status in the traversal by setting the mark to: *O* if the node is unmarked, i.e., the traversal hasn't encountered the node; *L* if the node has been traversed once towards the left subgraph; *R* if the traversal of the left subgraph has completed, and the traversal of the right subgraph began; *X* if both subgraphs have been traversed. Thus, we proceed to use graphs of type *binary-graph* $\{O, L, R, X\}$.

Along with modifying the nodes' marks, the edges of the graph are modified during traversal to keep the backtracking information. This is illustrated in Fig. 5, where γ_0 is the original unmarked graph, and γ is an intermediate graph halfway through the execution. The figure illustrates the traversal that started at n_1, proceeded to n_2, fully marked the left subgraph rooted at n_2, reached the node $p = n_5$ and is just about to traverse the left subgraph of p starting from the node $t = n_6$. The variables t (tip) and p (predecessor) are modified as the traversal advances. The idea of Schorr-Waite is that once the traversal has obtained the node t from which to proceed, the corresponding edge

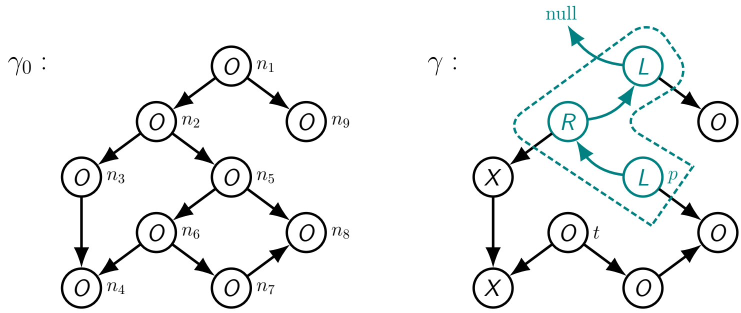

Fig. 5. The unmarked graph γ_0 with nodes $n_1, \ldots, n_9$, named in depth-first, left-to-right traversal order. The partially marked graph γ shows an intermediate state, with t the current tip node, and p its predecessor. The dashed line encloses the traversal stack (top = p), which contains exactly the nodes marked L or R. A node marked L (resp. R) has its left (resp. right) edge redirected to the predecessor through which it was reached. This inverts the edges on the stack: the path $n_1 \longrightarrow n_2 \longrightarrow n_5$ in γ_0 becomes $n_5 \longrightarrow n_2 \longrightarrow n_1$ in γ.

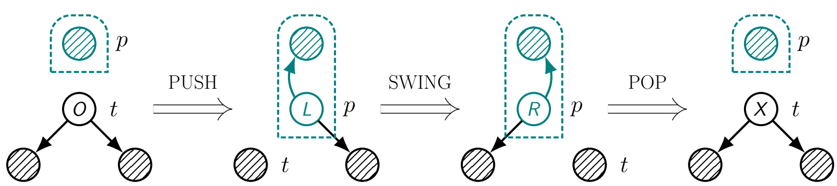

Fig. 6. Operations PUSH, SWING, and POP on the pivot node (unshaded), coordinate the marking of the pivot with the stack (dashed line) and with edge inversion. SWING and POP require that t is marked or null.

(or pointer) from p to t can temporarily be repurposed and redirected towards p's predecessor in the traversal (here n_2). As similar repurposing has been carried out for n_2 and n_1 when they were first encountered, this explicitly inverts in γ the path encompassing the sequence of nodes $\alpha = [n_1, n_2, n_5]$ in γ_0 (dashed line in Fig. 5). The sequence α serves the same role as the call stack in a recursive implementation, as it records the nodes whose subgraphs are currently being traversed, and the relative order in which each node has been reached. We refer to α as *the stack*, with α's *last* element (also stored in p) being the stack's top. A node on the stack can be marked L or R, but not O or X, as the latter signifies that the node's traversal hasn't started, or has finished, respectively. Conversely, a node marked L or R must be on the stack. These properties constitute some of the main invariants of the algorithm and are formalized in Section 5.

Fig. 6 zooms onto the tip node t (unshaded node, pivot) to illustrate how the traversal coordinates the marking and edge redirection around t with the modifications of t and p in three separate operations: PUSH, SWING, and POP. When t is first encountered, it's unmarked. PUSH promptly marks it L, and pushes it onto the stack (dashed line in Fig. 6) by redirecting its left edge towards p. The traversal continues by advancing t towards the left subgraph, and p to what t previously was. Once the left subgraph is fully marked, SWING restores pivot's left pointer, but keeps the pivot enlinked onto the stack by using the pivot's right pointer. The tip is swung to the right subgraph, and the mark changed to R, to indicate that the left subgraph has been traversed, and we're moving to the right subgraph. Once the pivot's right subgraph is traversed as well, POP sets the mark to X and restores the right edge. This returns the edges of the pivot to their originals from the initial graph, but also unlinks (i.e., pops) the pivot from the stack. Nodes stored into p and t are correspondingly shifted up, and the marking cycle continues from the new top of the stack.

```
t := r;  p := null;
if t = null then tm := true else tmp := t.m;  tm := (tmp ≠ O) end if;          // is t marked?
while p ≠ null ∨ ¬tm do
  if tm then
    pm := p.m;                                                                   // read p's mark
    if pm = R then
      tmp := p.r;  p.r := t;  p.m := X;  t := p;  p := tmp                      // POP
    else
      tmp1 := p.r;  tmp2 := p.l;  p.r := tmp2;  p.l := t;  p.m := R;  t := tmp1  // SWING
    end if
  else
    tmp := t.l;  t.l := p;  t.m := L;  p := t;  t := tmp                        // PUSH
  end if;
  if t = null then tm := true else tmp := t.m;  tm := (tmp ≠ O) end if         // is t marked?
end while
```

Fig. 7. Schorr-Waite algorithm. The algorithm follows the monadic style common to many separation logics, strictly dividing stateful *commands* from pure *expressions*. This requires pointer-dependent conditions (in if/while) to first dereference into variables, since expressions can't contain commands. For example, *tm* is assigned before and at the loop's end to enable its use in the loop condition.

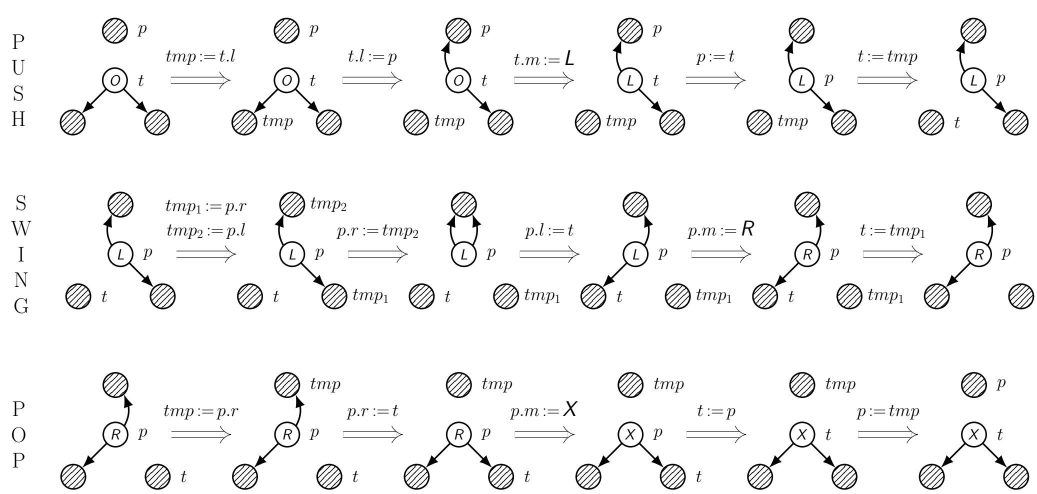

Fig. 8. Details of the main operations of Schorr-Waite.

Algorithm in Fig. 7 takes the root node r from which the traversal begins, and starts by setting the tip t to r, and p to null. An invariant of the algorithm is that p is always the top of the stack. Thus, p equals null if the stack is empty, otherwise p is marked L or R, but never O or X. Next, t's marking status is computed into *tm*. In general, if t is unmarked and non-null, then t is encountered for the first time. At each iteration, the algorithm mutates the graph—illustrated in Fig. 8—using: (1) PUSH, if t is unmarked and non-null, thus encountered for the first time, and can be pushed onto the stack; (2) SWING, if t is marked or null (encountered before), and p isn't marked R (thus, *is* marked L) signifying that p's left subgraph has been traversed; (3) POP, if t is marked or null and p is marked R signifying that p's right subgraph has been traversed. The algorithm terminates when p is null (the stack is empty) and t is marked or null (there's nothing to push onto the stack).

$$
\begin{aligned}
\mathit{inv}'\ \gamma_0\ \gamma\ \alpha\ t\ p\ \mathrel{\widehat{=}}\ & \mathit{uniq}\ (\text{null} \bullet \alpha) \land p = \mathit{last}\ (\text{null} \bullet \alpha) \land t \in \mathit{nodes}_0\ \gamma\ \land && (a)\\
& \mathit{closed}\ \gamma\ \land && (b)\\
& \mathit{nodes}\ \gamma/_{L,R} = \alpha\ \land && (c)\\
& \mathit{inset}\ \alpha\ \gamma = |\gamma|\ \land && (d)\\
& \mathit{restore}\ \alpha\ t\ \gamma = |\gamma_0|\ \land && (e)\\
& \mathit{nodes}\ \gamma/_O \subseteq \bigcup_{\alpha} (\mathit{reach}\ (\gamma/_O) \circ \gamma_r) \cup \mathit{reach}\ (\gamma/_O)\ t && (f)\\
\mathit{inv}\ \gamma_0\ \gamma\ t\ p\ \mathrel{\widehat{=}}\ & \exists \alpha.\ \mathit{inv}'\ \gamma_0\ \gamma\ \alpha\ t\ p
\end{aligned}
$$

Fig. 9. Main invariants of Schorr-Waite, formally. The parameter γ_0 is the initial graph, γ is the current partially marked graph, α, t and p are the current stack, tip and predecessor, respectively.

The verification task is to prove that the graph obtained upon termination modifies the initial unmarked graph as follows: (1) the nodes reachable from r, and only those, are marked X, and (2) the edges are restored to their initial versions.

5 Schorr-Waite's Invariants

To formalize the above specification, the first step is to state the main invariants of Schorr-Waite that relate the initial graph γ_0 to the current graph γ, the stack α, and the nodes t and p. In this formalization, we rely solely on combining graph abstractions from the small vocabulary introduced in Section 3. This ensures that later proofs remain compatible with graph decomposition, and depend only on general lemmas over this core vocabulary.

The relations to be captured are the following, all observed in the example graphs from Fig. 5. These invariants hold throughout the execution of Schorr-Waite, *except* inside the three main operations, PUSH, SWING, and POP, when the invariants are temporarily invalidated, to be restored by the time the operation terminates.

- (a) The stack α is a sequence of distinct nodes (i.e., each node is unique), also distinct from null, with p the top of stack (i.e., p is the last element of α, or null if α is empty). The tip t, being a child of p, is a node in γ, or null if the child doesn't exist.
- (b) The graph γ is closed, i.e., it has no dangling edges.
- (c) The stack α contains exactly the nodes that are partially marked (labeled L or R). Intuitively, this holds because α implements the call stack, and thus only contains the nodes whose subtrees are being currently marked.
- (d) The stack α describes a path in γ that *respects the markings*, in the following sense: if a node in α is marked L (resp. R), then its left (resp. right) child in γ is the node's predecessor in the traversal, and thus the node's predecessor in α. For example, in the graph γ in Fig. 5, the node n_5 is marked L and its left child n_2 is its predecessor in α. In other words, α stores the nodes in the relative order in which they are reached.
- (e) Reorienting the edges of the nodes in α produces the original graph γ_0 (modulo node marks).
- (f) The unmarked nodes (labeled O) are "to the right" of α and t, because the algorithm implements a left-to-right traversal order. More precisely, each unmarked node is reachable, by an unmarked path, either from t or from a right child of some node in α. For example, in the graph γ in Fig. 5, the nodes n_7, n_8, n_9 (unmarked) are "to the right" of α and t, whereas the nodes n_3, n_4 (marked X) are "to the left".

Fig. 9 presents the formal statements of the above invariants, which we proceed to explain. The encoding of (*a*) and (*b*) in Fig. 9 is direct. The statement (*c*) says that the stack α, viewed as a set rather than a sequence, equals the set of nodes in γ that are marked L or R. Statement (*f*) denotes by $\gamma_r\ x$ the right child of x in γ,[4] and directly says that an unmarked node x (i.e., $x \in \mathit{nodes}\ \gamma/_O$) is reachable from t by an unmarked path ($x \in \mathit{reach}\ (\gamma/_O)\ t$), or there exists a node $y \in \alpha$ such that x is reachable from the right child of y, also by an unmarked path ($x \in \bigcup_\alpha\ (\mathit{reach}\ (\gamma/_O) \circ \gamma_r)$). This leaves us with the statements (*d*) and (*e*), which we discuss next.

If one ignores for a moment the property of respecting the markings, then the English description of (*d*) and (*e*) says that the stack α is the contents of a linked list embedded in the graph γ, and that reversing the linkage of this list produces the original graph γ_0. Thus, one might consider following the approach to linked lists described in Section 2, and attempt to relate α with the *heap layouts* of γ and γ_0, by somehow relating $\mathit{list}\,\alpha$ with $\mathit{graph}\,\gamma_0$ and $\mathit{list}\,(\mathit{reverse}\,\alpha)$ with $\mathit{graph}\,\gamma$.[5] However, in our setting there is a more direct option, that elides the detour through heaps, and relates α to γ and γ_0 by means of the following graph transformation.

In particular, we first define the *higher-order* function *if-mark* that branches on the marking of an individual node, to output a transformation of the node's edges.

$$\begin{aligned}\mathit{if\text{-}mark} \quad &: \quad (\mathit{node} \to \mathit{node}) \to \mathit{node} \to \\ &\qquad\qquad \{O, L, R, X\} \times (\mathit{node} \times \mathit{node}) \to \mathit{node} \times \mathit{node}\end{aligned}$$

$$\mathit{if\text{-}mark}\ f\ x\ (m, [l, r]) \quad \widehat{=} \quad \begin{cases} [f\ x,\ r] & \text{if } m = L \\ [l,\ f\ x] & \text{if } m = R \\ [l, r] & \text{otherwise} \end{cases}$$

The transformation that *if-mark* computes is guided by the argument function f, itself mapping nodes to nodes. Thus, supplying different values for f produces different transformations, but within the general pattern encoded by *if-mark*. In more detail, *if-mark* applies to a node x of γ, x's mark m and children $[l, r]$ as follows. If m is L (resp. R), then x's left (resp. right) child is replaced in the output by $f\ x$, while the right (resp. left) child is passed along unchanged. Otherwise, if x is marked O or X, its children are returned unmodified.

We then apply the *map* combinator to *if-mark* in two different ways, to define the following two functions that will help us formalize the invariants (*d*) and (*e*) in a uniform way.

$$\begin{aligned}\mathit{inset} \ &:\ \mathit{seq\ node} \to \mathit{binary\text{-}graph}\ \{O, L, R, X\} \to \mathit{binary\text{-}graph\ unit} \\ \mathit{inset}\ \alpha\ \gamma \ &\widehat{=}\ \mathit{map}\ (\mathit{if\text{-}mark}\ (\mathit{prev}\ (\text{null} \bullet \alpha)))\ \gamma\end{aligned} \tag{14}$$

$$\begin{aligned}\mathit{restore} \ &:\ \mathit{node} \to \mathit{seq\ node} \to \mathit{binary\text{-}graph}\ \{O, L, R, X\} \to \mathit{binary\text{-}graph\ unit} \\ \mathit{restore}\ \alpha\ t\ \gamma \ &\widehat{=}\ \mathit{map}\ (\mathit{if\text{-}mark}\ (\mathit{next}\ (\alpha \bullet t)))\ \gamma\end{aligned} \tag{15}$$

Here $\mathit{prev}\ (\text{null} \bullet \alpha)$ is a function that takes a node x and returns the predecessor of the first occurrence of x in $(\text{null} \bullet \alpha)$ if $x \in \alpha$, or null if $x \notin \alpha$. Similarly, $\mathit{next}\ (\alpha \bullet t)\ x$ is the successor of the first occurrence of x in $(\alpha \bullet t)$ if $x \in \alpha$, or t if $x \notin \alpha$. For example, if α is the sequence $[n_1, n_2, n_5]$, then $\mathit{prev}\ (\text{null} \bullet \alpha)\ n_2 = n_1$, $\mathit{prev}\ (\text{null} \bullet \alpha)\ n_1 = \text{null}$, $\mathit{next}\ (\alpha \bullet t)\ n_2 = n_5$, and $\mathit{next}\ (\alpha \bullet t)\ n_5 = t$.

From the definition, it follows that graphs $\mathit{inset}\ \alpha\ \gamma$ and $\mathit{restore}\ \alpha\ t\ \gamma$ modify the graph γ by manipulating the edges related to the stack α, and otherwise erasing γ's marks. Specifically, $\mathit{inset}\ \alpha\ \gamma$ replaces the child (left/right, based on mark) of each node in α with its predecessor in $\text{null} \bullet \alpha$. For α to respect the markings, as required by the invariant (*d*), this transformation must actually preserve the graph. Thus, invariant (*d*) is formally stated in Fig. 9 as $\mathit{inset}\ \alpha\ \gamma = |\gamma|$. Similarly, $\mathit{restore}\ \alpha\ t\ \gamma$

[4] Dually, $\gamma_l\ x$ is the left child of x, so that $\gamma_{\text{adj}}\ x$ is the sequence $[\gamma_l\ x, \gamma_r\ x]$.

[5] In fact, relating α to the heap layouts of γ and γ_0 via the *list* predicate is precisely the approach of Yang's proof.

replaces the child (left/right, based on mark) of each node in α with its successor in $\alpha \bullet t$, thereby inverting the path α in γ, and redirecting the appropriate child of p towards t. Thus, invariant (e) is formally stated in Fig. 9 as *restore* $\alpha\ t\ \gamma = |\gamma_0|$.

Example. Consider the graphs γ and γ_0 from Fig. 5, taking $p = n_5$, $t = n_6$, and $\alpha = [n_1, n_2, n_5]$. The following calculation illustrates that *restore* $\alpha\ t\ \gamma$ computes the erasure of γ_0.

$$
\begin{aligned}
\mathit{restore}\ \alpha\ t\ \gamma &= \mathit{map}\ (\mathit{if\text{-}mark}\ (\mathit{next}\ (\alpha \bullet t)))\ \gamma \\
&= \quad n_1 \mapsto \mathit{if\text{-}mark}\ (\mathit{next}\ (\alpha \bullet t))\ n_1\ (L, [\mathrm{null}, n_9]) \\
&\quad \bullet n_2 \mapsto \mathit{if\text{-}mark}\ (\mathit{next}\ (\alpha \bullet t))\ n_2\ (R, [n_3, n_1]) \\
&\quad \bullet n_5 \mapsto \mathit{if\text{-}mark}\ (\mathit{next}\ (\alpha \bullet t))\ n_5\ (L, [n_2, n_8]) \\
&\quad \bullet \mathit{map}\ (\mathit{if\text{-}mark}\ (\mathit{next}\ (\alpha \bullet t)))\ (\gamma \backslash \{n_1, n_2, n_5\}) \\
&= \quad n_1 \mapsto [\mathit{next}\ (\alpha \bullet t)\ n_1, n_9] \\
&\quad \bullet n_2 \mapsto [n_3, \mathit{next}\ (\alpha \bullet t)\ n_2] \\
&\quad \bullet n_5 \mapsto [\mathit{next}\ (\alpha \bullet t)\ n_5, n_8] \\
&\quad \bullet |\gamma \backslash \{n_1, n_2, n_5\}| \\
&= n_1 \mapsto [n_2, n_9] \bullet n_2 \mapsto [n_3, n_5] \bullet n_5 \mapsto [t, n_8] \bullet |\gamma \backslash \{n_1, n_2, n_5\}| \\
&= |\gamma_0|
\end{aligned}
$$

When applied to $\gamma \backslash \{n_1, n_2, n_5\}$, the mapping returns the erasure $|\gamma \backslash \{n_1, n_2, n_5\}|$, because *if-mark* elides the contents, and doesn't modify the children of the nodes that aren't marked L or R. However, the calculations modifies the edges out of n_1, n_2 and n_5 in γ to obtain precisely the edges of γ_0.

On the other hand, the role of *inset* is to ensure that α is a marking-respecting path in γ, and thus a valid stack. For the same values of α and γ as above, *inset* $\alpha\ \gamma$ returns $|\gamma|$, because the passed α is indeed the stack in γ. However, if we passed a permutation of α, the equality to $|\gamma|$ won't hold. For example, consider passing $[n_2, n_5, n_1]$ for α.

$$
\begin{aligned}
\mathit{inset}\ \alpha\ \gamma &= \mathit{map}\ (\mathit{if\text{-}mark}\ (\mathit{prev}\ (\mathrm{null} \bullet \alpha)))\ \gamma \\
&= \quad n_1 \mapsto \mathit{if\text{-}mark}\ (\mathit{prev}\ (\mathrm{null} \bullet \alpha))\ n_1\ (L, [\mathrm{null}, n_9]) \\
&\quad \bullet n_2 \mapsto \mathit{if\text{-}mark}\ (\mathit{prev}\ (\mathrm{null} \bullet \alpha))\ n_2\ (R, [n_3, n_1]) \\
&\quad \bullet n_5 \mapsto \mathit{if\text{-}mark}\ (\mathit{prev}\ (\mathrm{null} \bullet \alpha))\ n_5\ (L, [n_2, n_8]) \\
&\quad \bullet \mathit{map}\ (\mathit{if\text{-}mark}\ (\mathit{prev}\ (\mathrm{null} \bullet \alpha)))\ (\gamma \backslash \{n_1, n_2, n_5\}) \\
&= \quad n_1 \mapsto [\mathit{prev}\ (\mathrm{null} \bullet \alpha)\ n_1, n_9] \\
&\quad \bullet n_2 \mapsto [n_3, \mathit{prev}\ (\mathrm{null} \bullet \alpha)\ n_2] \\
&\quad \bullet n_5 \mapsto [\mathit{prev}\ (\mathrm{null} \bullet \alpha)\ n_5, n_8] \\
&\quad \bullet |\gamma \backslash \{n_1, n_2, n_5\}| \\
&= n_1 \mapsto [n_5, n_9] \bullet n_2 \mapsto [n_3, \mathrm{null}] \bullet n_5 \mapsto [n_2, n_8] \bullet |\gamma \backslash \{n_1, n_2, n_5\}| \\
&\neq |\gamma|
\end{aligned}
$$

The calculation considers the nodes n_1, n_2 and n_5, as these are the nodes marked L or R. However, because the nodes aren't in the traversed order, the computation fails to encode $|\gamma|$. □

The significance of using the combinator *map* to define *inset* and *restore* is that the general lemmas from Section 3 apply, immediately deriving that both *inset* and *restore* are morphisms in the argument γ, and thus facilitating contextual localization in the proofs in Section 6. Similarly, using a higher-order function *if-mark* lets us prove general lemmas about it once, and use them several times. For example, Lemma 5.1 below states a property about *if-mark* that immediately applies to both *inset* and *restore*. Similarly, Lemma 5.2 is used to prove both equations (16) and (17) of Lemma 5.3, which are in turn each used several times in the proofs of the various components of Schorr-Waite in Section 6.

Lemma 5.1. *If* $nodes\ \gamma/_{L,R} = \emptyset$ *(i.e.,* γ *has no partially marked nodes) then* $map\ (\textit{if-mark}\ f)\ \gamma = |\gamma|$ *for any* f*. In particular,* $inset\ \alpha\ \gamma = restore\ \alpha\ t\ \gamma = |\gamma|$ *for any* α *and* t*.*

Lemma 5.2. *If* $f_1\ x = f_2\ x$ *for every* $x \in nodes\ \gamma/_{L,R}$ *then* $map\ (\textit{if-mark}\ f_1)\ \gamma = map\ (\textit{if-mark}\ f_2)\ \gamma$*.*

Lemma 5.3. *Let* α *be a sequence containing the nodes of* $\gamma/_{L,R}$*. Then the following equations hold.*

$$inset\ (\alpha \bullet p)\ \gamma = inset\ \alpha\ \gamma \tag{16}$$

$$restore\ (\alpha \bullet p)\ t\ \gamma = restore\ \alpha\ p\ \gamma \tag{17}$$

Lemma 5.1 holds because *if-mark* modifies only partially marked nodes, and applies erasure to the rest of the graph.

In Lemma 5.2, if f_1 and f_2 agree on partially marked nodes, then by definition, $\textit{if-mark}\ f_1\ x\ (\gamma\ x) = \textit{if-mark}\ f_2\ x\ (\gamma\ x)$ for all $x \in nodes\ \gamma$, because *if-mark* modifies only partially marked nodes. The conclusion then follows by Lemma 3.3.

Equation (16) follows from Lemma 5.2, by taking $f_1\ x = prev\ (\text{null} \bullet \alpha \bullet p)\ x$ and $f_2\ x = prev\ (\text{null} \bullet \alpha)\ x$, and observing that for every $x \in nodes\ \gamma/_{L,R}$ it must be $prev\ (\text{null} \bullet \alpha \bullet p)\ x = prev\ (\text{null} \bullet \alpha)\ x$, because $x \in \alpha$. Equation (17) follows similarly by taking $f_1\ x = next\ (\alpha \bullet p \bullet t)\ x$ and $f_2\ x = next\ (\alpha \bullet p)\ x$.

6 Proof of Schorr-Waite

The proof of the algorithm is divided into two steps. Following Yang [2001a], Section 6.1 first verifies Schorr-Waite assuming that the input graph is connected from the root node r. Section 6.2 verifies the POP fragment, and Section 6.3 extends to general graphs by framing the nodes *not* reachable from r. The subproof utilize diverse forms of contextual localization, as we discuss below.

6.1 Proof for Connected Graphs

We first establish the following specification.

$$\begin{array}{c}\{graph\ \gamma_0 \wedge closed\ \gamma_0 \wedge r \in nodes\ \gamma_0 \wedge nodes\ \gamma_0 = reach\ (\gamma_0/_O)\ r\} \\ \textit{Schorr-Waite}\ (r) \\ \{\exists \gamma.\ graph\ \gamma \wedge |\gamma_0| = |\gamma| \wedge \gamma = \gamma/_X\}\end{array} \tag{18}$$

The precondition in (18) says that the input graph γ_0 is well-formed ($graph\ \gamma_0$), closed, contains the root node r, and is unmarked and connected from r ($nodes\ \gamma_0 = reach\ (\gamma_0/_O)\ r$). The postcondition posits an ending graph γ which, aside from node marking, equals the input graph ($|\gamma_0| = |\gamma|$), and is fully marked itself ($\gamma = \gamma/_X$).

Fig. 10 presents the corresponding proof outline, most of which is self-explanatory by standard Hoare logic. For clarity, we elide the details about the stateful commands in lines 5, 15, 19, 24, and 28, summarizing their effect through assertions. The elided parts are shown elsewhere: for POP (line 15) in Section 6.2, for SWING (line 19), PUSH (line 24), line 5 and line 28 in the appendices. The proof of the remaining stateful command in line 11 is simple and elided altogether.

We detail the key non-trivial aspects of the proof outline: the non-spatial reasoning steps from the precondition at line 1 to line 2, and from line 31 to the postcondition at line 32. In both cases, the graph is at some point decomposed into disjoint parts (e.g., by Lemma 3.2 (2)), and the proof proceeds to analyze the relationship between those parts. These steps illustrate another example of

1. $\{\mathit{graph}\ \gamma_0 \wedge \mathit{closed}\ \gamma_0 \wedge r \in \mathit{nodes}\ \gamma_0 \wedge \mathit{nodes}\ \gamma_0 = \mathit{reach}\ (\gamma_0/_{O})\ r\}$
2. $\{\mathit{graph}\ \gamma_0 \wedge \mathit{inv}\ \gamma_0\ \gamma_0\ r\ \mathrm{null}\}$
3. $t := r;\ p := \mathrm{null};$
4. $\{\exists\gamma.\, \mathit{graph}\ \gamma \wedge \mathit{inv}\ \gamma_0\ \gamma\ t\ p\}$
5. **if** $t = \mathrm{null}$ **then** $\mathit{tm} := \mathrm{true}$ **else** $\mathit{tmp} := t.m;\ \mathit{tm} := (\mathit{tmp} \neq O)$ **end if**;
6. $\{\exists\gamma.\, \mathit{graph}\ \gamma \wedge \mathit{inv}\ \gamma_0\ \gamma\ t\ p \wedge \mathit{tm} = (t \in \mathit{marked}_0\ \gamma)\}$ // loop invariant
7. **while** $p \neq \mathrm{null} \vee \neg \mathit{tm}$ **do**
8. $\{\exists\gamma.\, \mathit{graph}\ \gamma \wedge \mathit{inv}\ \gamma_0\ \gamma\ t\ p \wedge \mathit{tm} = (t \in \mathit{marked}_0\ \gamma) \wedge (p \neq \mathrm{null} \vee \neg \mathit{tm})\}$
9. **if** tm **then**
10. $\{\exists\gamma.\, \mathit{graph}\ \gamma \wedge \mathit{inv}\ \gamma_0\ \gamma\ t\ p \wedge p \neq \mathrm{null} \wedge t \in \mathit{marked}_0\ \gamma\}$
11. $\mathit{pm} := p.m;$
12. $\{\exists\gamma.\, \mathit{graph}\ \gamma \wedge \mathit{inv}\ \gamma_0\ \gamma\ t\ p \wedge t \in \mathit{marked}_0\ \gamma \wedge \gamma_{\mathrm{val}}\ p = \mathit{pm} \in \{L, R\}\}$
13. **if** $\mathit{pm} = R$ **then**
14. $\{\exists\gamma.\, \mathit{graph}\ \gamma \wedge \mathit{inv}\ \gamma_0\ \gamma\ t\ p \wedge t \in \mathit{marked}_0\ \gamma \wedge \gamma_{\mathrm{val}}\ p = R\}$
15. $\mathit{tmp} := p.r;\ p.r := t;\ p.m := X;\ t := p;\ p := \mathit{tmp}$ // POP
16. $\{\exists\gamma.\, \mathit{graph}\ \gamma \wedge \mathit{inv}\ \gamma_0\ \gamma\ t\ p\}$
17. **else**
18. $\{\exists\gamma.\, \mathit{graph}\ \gamma \wedge \mathit{inv}\ \gamma_0\ \gamma\ t\ p \wedge t \in \mathit{marked}_0\ \gamma \wedge \gamma_{\mathrm{val}}\ p = L\}$
19. $\mathit{tmp}_1 := p.r;\ \mathit{tmp}_2 := p.l;\ p.r := \mathit{tmp}_2;\ p.l := t;\ p.m := R;\ t := \mathit{tmp}_1$ // SWING
20. $\{\exists\gamma.\, \mathit{graph}\ \gamma \wedge \mathit{inv}\ \gamma_0\ \gamma\ t\ p\}$
21. **end if**
22. **else**
23. $\{\exists\gamma.\, \mathit{graph}\ \gamma \wedge \mathit{inv}\ \gamma_0\ \gamma\ t\ p \wedge t \notin \mathit{marked}_0\ \gamma\}$
24. $\mathit{tmp} := t.l;\ t.l := p;\ t.m := L;\ p := t;\ t := \mathit{tmp}$ // PUSH
25. $\{\exists\gamma.\, \mathit{graph}\ \gamma \wedge \mathit{inv}\ \gamma_0\ \gamma\ t\ p\}$
26. **end if**;
27. $\{\exists\gamma.\, \mathit{graph}\ \gamma \wedge \mathit{inv}\ \gamma_0\ \gamma\ t\ p\}$
28. **if** $t = \mathrm{null}$ **then** $\mathit{tm} := \mathrm{true}$ **else** $\mathit{tmp} := t.m;\ \mathit{tm} := (\mathit{tmp} \neq O)$ **end if**
29. $\{\exists\gamma.\, \mathit{graph}\ \gamma \wedge \mathit{inv}\ \gamma_0\ \gamma\ t\ p \wedge \mathit{tm} = (t \in \mathit{marked}_0\ \gamma)\}$
30. **end while**
31. $\{\exists\gamma.\, \mathit{graph}\ \gamma \wedge \mathit{inv}\ \gamma_0\ \gamma\ t\ p \wedge p = \mathrm{null} \wedge t \in \mathit{marked}_0\ \gamma\}$
32. $\{\exists\gamma.\, \mathit{graph}\ \gamma \wedge |\gamma_0| = |\gamma| \wedge \gamma = \gamma/_{X}\}$

Fig. 10. Proof outline for the connected graph specification of Schorr-Waite. The fragments verifying the pointer primitives in lines 5, 11, 15, 19, 24, 28 are elided, and illustrated separately. Notation $\mathit{marked}_0\ \gamma$ abbreviates $\mathit{nodes}_0\ (\gamma/_{L,R,X})$, the set of marked nodes of γ, including null.

contextual localization, here within a proof outline that can be characterized as global, since all assertions in Fig. 10 refer to the full graph γ, rather than its parts.

Precondition (lines 1-2). Referring to Fig. 10, line 2 obtains from line 1 by unfolding the definition of *inv* from Fig. 9 with $t = r$, $p = \text{null}$, and instantiating the existentially quantified stack α with the empty sequence []. By setting $\alpha = []$, the unfolding derives proof obligations: $\mathit{uniq}\ \text{null}, \text{null} = \mathit{last}\ (\text{null} \bullet [])$, $r \in \mathit{nodes}_0\ \gamma_0$, $\mathit{closed}\ \gamma_0$, $\mathit{nodes}\ \gamma_0/_{L,R} = \emptyset$, $\mathit{inset}\ []\ \gamma_0 = |\gamma_0|$, $\mathit{restore}\ []\ t\ \gamma_0 = |\gamma_0|$, and $\mathit{nodes}\ \gamma_0/_O \subseteq \mathit{reach}\ (\gamma_0/_O)\ r$. The first four obligations are immediate, and properties $\mathit{inset}\ []\ \gamma_0 = |\gamma_0|$ and $\mathit{restore}\ []\ t\ \gamma_0 = |\gamma_0|$ follow by Lemma 5.1. The property $\mathit{nodes}\ \gamma_0/_O \subseteq \mathit{reach}\ (\gamma_0/_O)\ r$ is also easily shown because $\mathit{nodes}\ \gamma_0/_O \subseteq \mathit{nodes}\ \gamma_0$ and $\mathit{nodes}\ \gamma_0 = \mathit{reach}\ (\gamma_0/_O)\ r$ by the assumption in line 1.

To show the remaining $\mathit{nodes}\ \gamma_0/_{L,R} = \emptyset$, note that $\mathit{nodes}\ \gamma_0/_{L,R} \subseteq \mathit{nodes}\ \gamma_0$ and $\mathit{reach}\ (\gamma_0/_O)\ r \subseteq \mathit{nodes}\ \gamma_0/_O$ because filtering and reachability select a subset of graph's nodes. Thus, $\mathit{nodes}\ \gamma_0/_{L,R} \subseteq \mathit{nodes}\ \gamma_0 = \mathit{reach}\ (\gamma_0/_O)\ r \subseteq \mathit{nodes}\ \gamma_0/_O$. By Lemma 3.2 (2), $\gamma_0/_{L,R}$ and $\gamma_0/_O$ select node-disjoint subgraphs of γ_0. But $\mathit{nodes}\ \gamma_0/_{L,R}$ can be a subset of $\mathit{nodes}\ \gamma_0/_O$ with which it's disjoint, only if $\mathit{nodes}\ \gamma_0/_{L,R} = \emptyset$.

Postcondition (lines 31-32). From $\mathit{inv}\ \gamma_0\ \gamma\ t\ p$ in line 31, it follows $p = \mathit{last}\ (\text{null} \bullet \alpha)$ and $\mathit{uniq}\ (\text{null} \bullet \alpha)$. As $p = \text{null}$, it must be $\alpha = []$, i.e., the stack is empty. Then, by (*e*) in Fig. 9, $|\gamma_0| = \mathit{restore}\ []\ t\ \gamma$, which in turn equals $|\gamma|$ by Lemma 5.1, thus proving $|\gamma_0| = |\gamma|$ in the postcondition. By Lemma 3.2 (2), the remaining $\gamma = \gamma/_X$ from the postcondition is equivalent to $\mathit{nodes}\ \gamma/_{O,L,R} = \emptyset$. By Lemma 3.2 (2) again, and distributivity of *nodes*, the latter is further equivalent to $\mathit{nodes}\ \gamma/_O \cup \mathit{nodes}\ \gamma/_{L,R} = \emptyset$. By (*c*) in Fig. 9, $\mathit{nodes}\ \gamma/_{L,R} = \emptyset$, so it suffices to show $\mathit{nodes}\ \gamma/_O = \emptyset$. By (*f*) of Fig. 9, $\mathit{nodes}\ \gamma/_O \subseteq \mathit{reach}\ (\gamma/_O)\ t$, i.e., unmarked nodes of γ are reachable from t by unmarked paths. But t itself is marked in line 31, so no node is reachable from t by an unmarked path. Hence, $\mathit{reach}\ (\gamma/_O)\ t = \emptyset$, and thus $\mathit{nodes}\ \gamma/_O = \emptyset$.

6.2 Proof of POP

The pre- and postcondition for POP derive from lines 14 and 16 of Fig. 10.

$$\begin{array}{c}\{\exists \gamma.\ \mathit{graph}\ \gamma \wedge \mathit{inv}\ \gamma_0\ \gamma\ t\ p \wedge t \in \mathit{marked}_0\ \gamma \wedge \gamma_{\text{val}}\ p = R\} \\ \text{POP} \\ \{\exists \gamma'.\ \mathit{graph}\ \gamma' \wedge \mathit{inv}\ \gamma_0\ \gamma'\ t\ p\}\end{array} \tag{19}$$

The precondition says that the heap implements a well-formed graph γ, that satisfies the invariant, given t and p. Additionally, t is marked or null and p is marked R, as POP is invoked only if these properties are satisfied. The postcondition asserts that the heap represents a new graph γ' that satisfies the invariant for the updated values of t and p. The specification is given solely in terms of graphs to hide the internal low-level reasoning about pointers, and expose only the more abstract graph properties required in Fig. 10.

The proof outline is in Fig. 11 and divides into two parts. The first part (lines 1-9) serves to show that the stateful commands implementing POP are safe to execute, and result in a valid graph γ'. The second part (lines 9-10) serves to show that γ' actually satisfies the Schorr-Waite invariants. Along with the analogous steps in the proofs of SWING and PUSH, this is the most important, and most substantial part of the whole verification. We discuss the two parts in more detail.

POP produces valid graph (lines 1-9). Transitioning from line 1 to line 2 involves eliminating the existential quantifier and saving the current values of t, p, and the left and right child of p into t_0, p_0, p_l and p_r, respectively. Saving these values prepares for line 3 to omit the last three

1. $\{\exists \gamma.\, \mathit{graph}\ \gamma \wedge \mathit{inv}\ \gamma_0\ \gamma\ t\ p \wedge t \in \mathit{marked}_0\ \gamma \wedge \gamma_{\mathrm{val}}\ p = R\}$
2. $\quad \{\mathit{graph}\ \gamma \wedge t_0 = t \wedge p_0 = p$
   $\quad\quad \wedge\ \mathit{inv}\ \gamma_0\ \gamma\ t_0\ p_0 \wedge t_0 \in \mathit{marked}_0\ \gamma \wedge \gamma\ p_0 = (R, [p_l, p_r])\}$
3. $\quad\quad \{\mathit{graph}\ (p_0 \mapsto (R, [p_l, p_r]) \bullet \gamma \backslash p_0) \wedge t_0 = t \wedge p_0 = p\}$
4. $\quad\quad \{(p_0 \Mapsto R, p_l, p_r \wedge t_0 = t \wedge p_0 = p) * \mathit{graph}\ \gamma \backslash p_0\}$
5. $\quad\quad\quad \{p_0 \Mapsto R, p_l, p_r \wedge t_0 = t \wedge p_0 = p\}$
   $\quad\quad\quad \mathit{tmp} := p.r;\ p.r := t;\ p.m := X;\ t := p;\ p := \mathit{tmp};$ // POP
6. $\quad\quad\quad \{p_0 \Mapsto X, p_l, t_0 \wedge t = p_0 \wedge p = p_r\}$
7. $\quad\quad \{(p_0 \Mapsto X, p_l, t_0 \wedge t = p_0 \wedge p = p_r) * \mathit{graph}\ \gamma \backslash p_0\}$
8. $\quad\quad \{\mathit{graph}\ (p_0 \mapsto (X, [p_l, t_0]) \bullet \gamma \backslash p_0) \wedge t = p_0 \wedge p = p_r\}$
9. $\quad \{\mathit{graph}\ (p_0 \mapsto (X, [p_l, t_0]) \bullet \gamma \backslash p_0) \wedge t = p_0 \wedge p = p_r$
   $\quad\quad \wedge\ \mathit{inv}\ \gamma_0\ \gamma\ t_0\ p_0 \wedge t_0 \in \mathit{marked}_0\ \gamma \wedge \gamma\ p_0 = (R, [p_l, p_r])\}$
10. $\{\exists \gamma'.\, \mathit{graph}\ \gamma' \wedge \mathit{inv}\ \gamma_0\ \gamma'\ t\ p\}$

Fig. 11. Proof outline for POP. The property $\mathit{inv}\ \gamma_0\ \gamma\ t_0\ p_0 \wedge t_0 \in \mathit{marked}_0\ \gamma \wedge \gamma\ p_0 = (R, [p_l, p_r])$ propagates from line 2 to line 9 because it doesn't describe the heap or variables mutated by the program.

conjuncts of line 2, which will be re-attached in line 9. The move is valid, because the conjuncts are unaffected by execution, as they don't involve the heap or the variables mutated by POP.

Proceeding with the proof outline, line 3 expands γ around p_0 (=p), using equation (5), so that distributivity of *graph* (4) applies in line 4 to frame away $\mathit{graph}\ \gamma \backslash p_0$. This is similar to how the pointer j in Section 2 was separated from the rest of the heap, in order to verify the command that mutates it. The derivation of line 6 is elided because it's standard, involving several applications of framing and the inference rules for the stateful primitives. It suffices to say that the line reflects the mutations of the pointer p_0, and the assignment of new values to t and p, in accord with the illustration of POP in Fig. 6. In particular, the right edge of p_0 is redirected towards t_0, the node is marked X, and t and p are set to p_0 and p_r, respectively. The remainder of the proof outline up to line 9 restores the framed graph and the non-spatial conjuncts omitted in line 3.

Invariant preservation (lines 9-10). This part of the proof is fully non-spatial and relies almost entirely on the distributivity of various morphisms, along with a few general graph lemmas from Section 3. It follows the same pattern of contextual locality as Lemma 3.6, exploiting that the graphs in lines 9 and 10 share the subgraph $\gamma \backslash p_0$.

As the first step, we reformulate the problem as the following implication, whose premise restates the assertions about γ from line 9, while the conclusion specifies the ending graph γ'.

$$\gamma = p_0 \mapsto (R, [p_l, p_r]) \bullet \gamma \backslash p_0 \wedge \tag{20}$$

$$\mathit{inv}'\ \gamma_0\ \gamma\ (\alpha \bullet p_0)\ t_0\ p_0 \wedge \tag{21}$$

$$t_0 \in \mathit{marked}_0\ \gamma \implies \tag{22}$$

$$\exists \gamma'.\, \gamma' = p_0 \mapsto (X, [p_l, t_0]) \bullet \gamma \backslash p_0 \wedge \tag{23}$$

$$\mathit{inv}'\ \gamma_0\ \gamma'\ \alpha\ p_0\ p_r \tag{24}$$

(c) $nodes\ \gamma'/_{L,R} =$	Def. of γ'
$= nodes\ (p_0 \mapsto (X, [p_l, t_0]) \bullet \gamma \backslash p_0)/_{L,R}$	Lem. 3.1 (1) & 3.1 (3) (distrib.)
$= nodes\ (p_0 \mapsto (X, [p_l, t_0]))/_{L,R} \uplus nodes\ (\gamma \backslash p_0)/_{L,R}$	Def. of *filter* (10)
$= nodes\ (\gamma \backslash p_0)/_{L,R}$	Assump. (21.c)
$= \alpha$	
(d) $inset\ \alpha\ \gamma' =$	Def. of γ'
$= inset\ \alpha\ (p_0 \mapsto (X, [p_l, t_0]) \bullet \gamma \backslash p_0)$	Lem. 3.1 (5) (distrib.)
$= inset\ \alpha\ (p_0 \mapsto (X, [p_l, t_0])) \bullet inset\ \alpha\ \gamma \backslash p_0$	Def. of *inset* (14)
$= p_0 \mapsto [p_l, t_0] \bullet inset\ \alpha\ \gamma \backslash p_0$	Lem. 5.3
$= p_0 \mapsto [p_l, t_0] \bullet inset\ (\alpha \bullet p_0)\ \gamma \backslash p_0$	Assump. (21.d)
$= p_0 \mapsto [p_l, t_0] \bullet \lvert \gamma \backslash p_0 \rvert$	Def. of *erasure* (8)
$= \lvert p_0 \mapsto (X, [p_l, t_0]) \rvert \bullet \lvert \gamma \backslash p_0 \rvert$	Lem. 3.1 (4) (distrib.)
$= \lvert p_0 \mapsto (X, [p_l, t_0]) \bullet \gamma \backslash p_0 \rvert$	Def. of γ'
$= \lvert \gamma' \rvert$	
(e) $restore\ \alpha\ p_0\ \gamma' =$	Def. of γ'
$= restore\ \alpha\ p_0\ (p_0 \mapsto (X, [p_l, t_0]) \bullet \gamma \backslash p_0)$	Lem. 3.1 (5) (distrib.)
$= restore\ \alpha\ p_0\ (p_0 \mapsto (X, [p_l, t_0])) \bullet restore\ \alpha\ p_0\ \gamma \backslash p_0$	Def. of *restore* (15)
$= p_0 \mapsto [p_l, t_0] \bullet restore\ \alpha\ p_0\ \gamma \backslash p_0$	Lem. 5.3
$= p_0 \mapsto [p_l, t_0] \bullet restore\ (\alpha \bullet p_0)\ t_0\ \gamma \backslash p_0$	Def. of *restore* (15)
$= restore\ (\alpha \bullet p_0)\ t_0\ (p_0 \mapsto (R, [p_l, p_r]))$	
$\quad \bullet\ restore\ (\alpha \bullet p_0)\ t_0\ \gamma \backslash p_0$	Lem. 3.1 (5) (distrib.)
$= restore\ (\alpha \bullet p_0)\ t_0\ (p_0 \mapsto (R, [p_l, p_r]) \bullet \gamma \backslash p_0)$	Assump. (21.e)
$= \lvert \gamma_0 \rvert$	
(f) $nodes\ \gamma'/_O =$	Def. of *filter*, *nodes*, γ & γ'
$= nodes\ \gamma/_O$	Assump. (21.f)
$\subseteq \bigcup_{\alpha \cdot p_0} (reach\ (\gamma/_O) \circ \gamma_r) \cup reach\ (\gamma/_O)\ t_0$	$t_0 \notin nodes\ \gamma/_O$
$= \bigcup_{\alpha \cdot p_0} (reach\ (\gamma/_O) \circ \gamma_r)$	Comm.&Assoc. of $\cup$
$= \bigcup_{\alpha} (reach\ (\gamma/_O) \circ \gamma_r) \cup reach\ (\gamma/_O)\ (\gamma_r\ p_0)$	$\gamma_r\ p_0 \notin nodes\ \gamma/_O$
$= \bigcup_{\alpha} (reach\ (\gamma/_O) \circ \gamma_r)$	$(\gamma_r = \gamma'_r$ on $\alpha)$ & $(\gamma/_O = \gamma'/_O)$
$= \bigcup_{\alpha} (reach\ (\gamma'/_O) \circ \gamma'_r)$	$p_0 \notin nodes\ \gamma'/_O$
$= \bigcup_{\alpha} (reach\ (\gamma'/_O) \circ \gamma'_r) \cup reach\ (\gamma'/_O)\ p_0$	

Fig. 12. POP preserves invariants (c)-(f) for $\gamma = p_0 \mapsto (R, [p_l, p_r]) \bullet \gamma \backslash p_0$ and $\gamma' = p_0 \mapsto (X, [p_l, t_0]) \bullet \gamma \backslash p_0$.

To see that the premise follows from line 9, note that (20) expands γ around p_0, using $\gamma\ p_0 = (R, [p_l, p_r])$ from line 9; that (21) holds because $inv\ \gamma_0\ \gamma\ t_0\ p_0$ in line 9 implies that p_0 is the top of stack of γ, which thus has the form $\alpha \bullet p_0$ for some α; and that (22) is an explicit conjunct in line 9.

In the conclusion of the implication, (23) reflects that the p_0 is now marked X, and that its right edge is restored towards t_0. On the other hand, (24) indicates that the new stack is α (node p_0 having been popped from the prior stack $\alpha \bullet p_0$), and the new tip and stack's top are p_0 and p_r respectively. These changes correspond to the illustration of POP in Fig. 8, if one takes p_0, t_0 and p_r as the initial values of p, t, and p's right child. It's also readily apparent that (23) and (24) imply the postcondition of POP in line 10.

1. $\{graph\ \gamma_0 \wedge closed\ \gamma_0 \wedge r \in nodes\ \gamma_0 \wedge \gamma_0 = \gamma_0/_O\}$
2. $\{\exists \gamma_1 \gamma_2.\ graph\ (\gamma_1 \bullet \gamma_2) \wedge closed\ \gamma_1 \wedge r \in nodes\ \gamma_1 \wedge nodes\ \gamma_1 = reach\ (\gamma_1/_O)\ r$
   $\wedge\ nodes\ \gamma_1 = reach\ ((\gamma_1 \bullet \gamma_2)/_O)\ r \wedge \gamma_2 = \gamma_2/_O \wedge \gamma_0 = \gamma_1 \bullet \gamma_2\}$
3. $\{(graph\ \gamma_1 \wedge closed\ \gamma_1 \wedge r \in nodes\ \gamma_1 \wedge nodes\ \gamma_1 = reach\ (\gamma_1/_O)\ r)$
   $*\ (graph\ \gamma_2 \wedge nodes\ \gamma_1 = reach\ ((\gamma_1 \bullet \gamma_2)/_O)\ r \wedge \gamma_2 = \gamma_2/_O \wedge \gamma_0 = \gamma_1 \bullet \gamma_2)\}$
4. $\{graph\ \gamma_1 \wedge closed\ \gamma_1 \wedge r \in nodes\ \gamma_1 \wedge nodes\ \gamma_1 = reach\ (\gamma_1/_O)\ r\}$
   Schorr-Waite (r)
5. $\{\exists \gamma_1'.\ graph\ \gamma_1' \wedge |\gamma_1| = |\gamma_1'| \wedge \gamma_1' = \gamma_1'/_X\}$
6. $\{(\exists \gamma_1'.\ graph\ \gamma_1' \wedge |\gamma_1| = |\gamma_1'| \wedge \gamma_1' = \gamma_1'/_X)$
   $*\ (graph\ \gamma_2 \wedge nodes\ \gamma_1 = reach\ ((\gamma_1 \bullet \gamma_2)/_O)\ r \wedge \gamma_2 = \gamma_2/_O \wedge \gamma_0 = \gamma_1 \bullet \gamma_2)\}$
7. $\{\exists \gamma_1'.\ graph(\gamma_1' \bullet \gamma_2) \wedge |\gamma_0| = |\gamma_1' \bullet \gamma_2|$
   $\wedge\ \gamma_1' \bullet \gamma_2 = (\gamma_1' \bullet \gamma_2)/_X \bullet (\gamma_1' \bullet \gamma_2)/_O \wedge nodes\ (\gamma_1' \bullet \gamma_2)/_X = reach\ (\gamma_0/_O)\ r\}$
8. $\{\exists \gamma.\ graph\ \gamma \wedge |\gamma_0| = |\gamma| \wedge \gamma = \gamma/_X \bullet \gamma/_O \wedge nodes\ \gamma/_X = reach\ (\gamma_0/_O)\ r\}$

Fig. 13. Proof outline for the general graph specification of Schorr-Waite.

The second step of the proof proceeds to establish the invariant $inv'\ \gamma_0\ \gamma'\ \alpha\ p_0\ p_r$ from (24) out of $inv'\ \gamma_0\ \gamma\ (\alpha \bullet p_0)\ t_0\ p_0$ in (21), and $t_0 \in marked_0\ \gamma$ from (22). As inv' is defined in Fig. 9 in terms of conjuncts (a)-(f), each conjunct in (24) is proved starting from the corresponding conjunct in (21). The subcase (b) of this proof is explicitly Lemma 3.6, and Fig. 12 presents the subcases (c)-(f) in the self-explanatory equational style. The proofs follow the pattern of contextual localization from Lemma 3.6. They begin by decomposing γ and γ' as in (20) and (23), isolating the node p_0 from the shared subcomponent $\gamma \backslash p_0$. Each subcase then applies the appropriate morphism to distribute over the decomposition, ultimately reducing the goal to a property of $\gamma \backslash p_0$ that is already assumed.

The remaining subcase (a) doesn't have an equational flavor so we give it here explicitly. That case requires showing that (20-22) imply $uniq\ (\text{null} \bullet \alpha)$, $p_r = last\ (\text{null} \bullet \alpha)$ and $p_0 \in nodes_0\ \gamma'$. The only non-trivial property is $p_r = last\ (\text{null} \bullet \alpha)$, i.e., that p_r is the top of the stack α. This holds because (21) gives $inset\ (\alpha \bullet p_0)\ \gamma = |\gamma|$, i.e., that the stack $\alpha \bullet p_0$ is a path in γ. By definition of $inset$ (14), $prev\ (\text{null} \bullet \alpha \bullet p_0)\ p_0 = \gamma_r\ p_0 = p_r$, and thus $p_r = last\ (\text{null} \bullet \alpha)$.

6.3 Proof for General Graphs

We can now establish the following general specification of Schorr-Waite, which doesn't assume that γ is connected from r.

$$\{graph\ \gamma_0 \wedge closed\ \gamma_0 \wedge r \in nodes\ \gamma_0 \wedge \gamma_0 = \gamma_0/_O\}$$
$$\textit{Schorr-Waite}\ (r)$$
$$\{\exists \gamma.\ graph\ \gamma \wedge |\gamma_0| = |\gamma| \wedge \gamma = \gamma/_X \bullet \gamma/_O \wedge nodes\ \gamma/_X = reach\ (\gamma_0/_O)\ r\}$$

As in specification (18), the precondition says that the input graph γ_0 is well-formed ($graph\ \gamma_0$), closed, contains the node r, and is unmarked ($\gamma_0 = \gamma_0/_O$), but elides the conjunct about connectedness. The postcondition posits an ending graph γ which, aside from node marking, equals the input graph ($|\gamma_0| = |\gamma|$), and splits into fully marked and unmarked parts ($\gamma = \gamma/_X \bullet \gamma/_O$), with the fully-marked part corresponding to the nodes initially reachable from r ($nodes\ \gamma/_X = reach\ (\gamma_0/_O)\ r$).

The proof is in Fig. 13, and is obtained generically, by framing (18) without reanalyzing the code. It utilizes distributivity of *graph* to set up the framing, and morphisms to propagate the framed information from smaller to larger graph. This propagation, carried out in the last step of the proof (from line 7 to line 8), is a form of contextual localization in reverse, where instead of focusing inward, information flows outward through the structure.

In more detail, γ_0 is first split disjointly into $\gamma_1 = \gamma_0/_{(reach\ \gamma_0\ r)}$ (part connected from r), and its complement γ_2, so that $\gamma_0 = \gamma_1 \bullet \gamma_2$. By definition then, $nodes\ \gamma_1 = reach\ (\gamma_1 \bullet \gamma_2)\ r$, so that by Lemma 3.5 (2), $closed\ \gamma_1$, $r \in nodes\ \gamma_1$ and $nodes\ \gamma_1 = reach\ \gamma_1\ r$. Because γ_1 and γ_2 are subgraphs of γ_0, they are also unmarked ($\gamma_1/_O = \gamma_1$ and $\gamma_2/_O = \gamma_2$), thus obtaining line 2 in Fig. 13. Distributivity of *graph* (4) then derives line 3, which is a separating conjunction of the precondition from (18) with the predicate that collects all the conjuncts from line 2 that refer to graph γ_2. By framing the specification (18), we can thus derive a postcondition for Schorr-Waite in line 6 that is a separating conjunction of the conjuncts about γ_2, with the postcondition from (18) that asserts the existence of a graph γ_1' such that $|\gamma_1| = |\gamma_1'|$ and $\gamma_1' = \gamma_1'/_X$.

Proceeding, line 7 joins the graphs γ_1' and γ_2 together as follows. First, distributivity of *graph* and erasure obtains $graph\ (\gamma_1' \bullet \gamma_2)$, and $|\gamma_0| = |\gamma_1 \bullet \gamma_2| = |\gamma_1' \bullet \gamma_2|$. We also get $\gamma_1'/_O = \gamma_1'/_X/_O = e$ and $\gamma_2/_X = \gamma_2/_O/_X = e$ by Lemma 3.2 (4). From here, $\gamma_1' \bullet \gamma_2 = \gamma_1'/_X \bullet \gamma_2/_O = (\gamma_1' \bullet \gamma_2)/_X \bullet (\gamma_1' \bullet \gamma_2)/_O$ and also $nodes\ (\gamma_1' \bullet \gamma_2)/_X = nodes\ \gamma_1'/_X = nodes\ \gamma_1' = nodes\ |\gamma_1'| = nodes\ |\gamma_1| = nodes\ \gamma_1 = reach\ (\gamma_0/_O)\ r$. Finally, line 8 abstracts over $\gamma = \gamma_1' \bullet \gamma_2$.

7 Sketch of Union-Find Verification

The union-find data structure manages a collection of disjoint sets, supporting efficient set merging and membership queries. Each set is represented in memory as an inverted tree, each element (a node in the tree) points to its unique parent, and the root of the tree—which points to itself—serves as the set's representative. An element's representative is obtained by following parent pointers upward. The union operation merges two sets by making the root of one point to the root of the other set, thereby unifying their representatives (Fig. 14).

Because each node has a unique parent, union-find operates over *unary graphs*, with a *graph* predicate analogous to that for binary graphs from Section 3.

$$\begin{aligned} graph_1\ e &\ \widehat{=}\ emp \\ graph_1\ (x \mapsto y \bullet \gamma) &\ \widehat{=}\ x \Mapsto y * graph_1\ \gamma \end{aligned}$$

The main graph primitives in the specification for union-find are the following.

$$\begin{aligned} summit\ \gamma\ x &\ \widehat{=}\ \begin{cases} \bigcup_{\gamma_{\text{adj}}\ x} summit\ (\gamma \backslash x) & \text{if } x \in nodes\ \gamma \\ \{x\} & \text{otherwise} \end{cases} \\ summits\ \gamma &\ \widehat{=}\ \bigcup_{nodes\ \gamma} summit\ \gamma \\ loops\ \gamma &\ \widehat{=}\ \{x \mid x \in \gamma_{\text{adj}}\ x\} \end{aligned}$$

The set $summit\ \gamma\ x$ contains the nodes of γ that *mark an end (i.e., summit)* of a path from x. More precisely, $z \in summit\ \gamma\ x$ if there exists a path from x to some node y in γ such that z is a child of y, and either $z \notin nodes\ \gamma$ (i.e., the edge from y to z is dangling) or z is already in the path from x to y (thus starting a cycle). The set $summits\ \gamma$ collects all such ending nodes. The set $loops\ \gamma$ collects the nodes of γ that include themselves in their adjacency list; that is, nodes forming cycles of size 1.

Fig. 14 illustrates the relevance of the above primitives for union-find. Specifically, *summit* computes the representative of a given node (e.g, $summit\ \gamma\ c = \{a\}$), as the representatives in union-find are precisely the path-ending nodes. Analogously, *summits* collects all the representatives (e.g.,

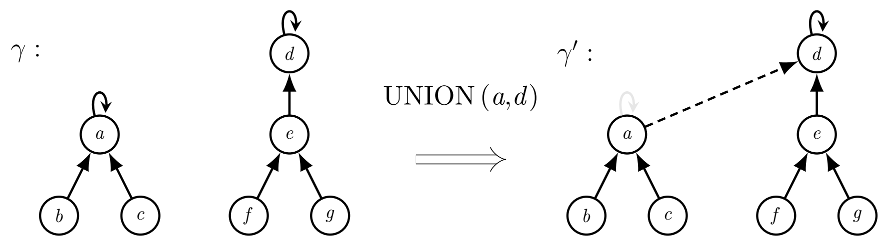

Fig. 14. Two disjoint sets encoded as inverted trees being merged by the operation of UNION. Initially, a and d represent the sets $\{a, b, c\}$ and $\{d, e, f, g\}$, respectively. After the union, d represents all the elements.

$\mathit{summits}\ \gamma = \{a, d\}$ and $\mathit{summits}\ \gamma' = \{d\}$). Importantly, *summit* and *summits* are somewhat more general still, as they may return nodes that are outside of the graph (dangling edges), or nodes that start cycles. In particular, if $\mathit{summits}\ \gamma$ only contains nodes in γ, then γ is closed. Furthermore, if $\mathit{summits}\ \gamma$ only contains nodes that form *trivial* cycles (i.e., of size 1, from $\mathit{loops}\ \gamma$), then γ is acyclic (modulo representatives), and thus an inverted tree.

We thus use the above primitives not only to compute the representatives, but also to define when a graph is an inverted tree. We do so via the following predicate $\mathit{set}\ S\ x$ which holds of a heap that contains the layout of a *single set* S in a union-find collection, whose representative is x.

$$\mathit{set}\ S\ x\ \widehat{=}\ \exists \gamma.\ \mathit{graph}_1\ \gamma \land \mathit{summits}\ \gamma = \mathit{loops}\ \gamma = \{x\} \land \mathit{nodes}\ \gamma = S$$

Indeed, the definition requires the existence of a unary graph γ whose nodes are exactly S, such that γ has x as the unique summit (i.e., γ is closed), and x forms a trivial cycle (i.e., γ is inverted tree). For example, in the graph γ in Fig. 14 we have $\mathit{set}\ \{a, b, c\}\ a$ and $\mathit{set}\ \{d, e, f, g\}\ d$.

The following Hoare triples specify the methods of union-find, where the method's postcondition denotes the method's return value by the dedicated variable *result*.[6]

$$\begin{array}{rcl}
\{\mathit{emp}\} & \text{NEW} & \{\mathit{set}\ \{\mathit{result}\}\ \mathit{result}\} \\
\{\mathit{set}\ S\ y \land x \in S\} & \text{FIND}\ (x) & \{\mathit{set}\ S\ y \land \mathit{result} = y\} \\
\{\mathit{set}\ S_1\ x_1 * \mathit{set}\ S_2\ x_2\} & \text{UNION}\ (x_1, x_2) & \{\mathit{set}\ (S_1 \uplus S_2)\ \mathit{result} \land \mathit{result} \in \{x_1, x_2\}\}
\end{array}$$

NEW starts from an empty heap and allocates a node that forms a singleton set and serves as its own representative. FIND takes a node x known to belong to some set S, and returns the representative y of S. UNION assumes that x_1 and x_2 are representatives of disjoint sets and joins the sets, returning one of x_1 or x_2 as the new representative (Fig. 14). Notably, our specifications refer only to the disjoint sets each operation manipulates, following the *small footprint* style. While this is the norm in separation logic generally, we're unaware of prior union-find verifications that adopt it.

The implementation and the proof outlines for the methods are in the appendix. Here, we only discuss how morphisms and distributivity help with the verification of union-find. The main challenge in this verification is establishing, in various subproofs, the inclusion $\mathit{summits}\ \gamma \subseteq \mathit{loops}\ \gamma$, which is one side of the defining equation $\mathit{summits}\ \gamma = \mathit{loops}\ \gamma$ from the *set* predicate. This closely resembles the definition of *closed* in Section 3 as $\mathit{sinks}\ \gamma \subseteq \mathit{nodes}_0\ \gamma$, where the fact that *sinks* and *nodes* are morphisms enabled contextual localization via Lemma 3.6.

In the case of union-find, we apply the same principle of localization by leveraging distribution properties of *loops* and *summits*. The former is a standard PCM morphism. The latter, however, exhibits more nuanced behavior. First, it distributes only under the specific condition that all cycles

[6]This requires extending the heretofore used standard logic of O'Hearn, Reynolds and Yang with value-returning methods, but Hoare Type Theory admits such an extension.

in the graph are trivial—a property satisfied by union-find graphs. Second, and more unusually, its distribution doesn't follow the standard additive form of morphisms, but *subtracts* nodes from the opposing component to avoid interference.

$$\mathit{summits}\ (\gamma_1 \bullet \gamma_2) = (\mathit{summits}\ \gamma_1) \backslash \mathit{nodes}\ \gamma_2\ \cup\ (\mathit{summits}\ \gamma_2) \backslash \mathit{nodes}\ \gamma_1$$

The intuition is that a summit in a subgraph γ_1 is a path-ending node. If the path ends with a dangling edge into γ_2, the summit ceases to be path-ending in the composition, where the path continues. The equation accounts for this by explicitly removing the overlap through set subtraction.

While the full theory of *loops* and *summits* is more extensive, and holds for general (not just unary) graphs, the distribution principle above suffices to draw a high-level analogy with Schorr-Waite, and illustrate how contextual localization applies equally well to union-find.

8 Related work

Proofs of Schorr-Waite in separation logic. The starting point of our paper has been Yang's proof [Yang 2001a,b], which is an early work on separation logic generally, and the first work on graphs in separation logic specifically. The distinction with our proof is that Yang doesn't use mathematical graphs as an explicit argument to the *graph* predicate, but rather relies on non-spatial proxy abstractions, such as the spanning tree of the graph, and various subsets of nodes. These proxy abstractions aren't independent, and to keep them synchronized with each other and with the graph laid out in the heap, the proof must frequently switch between non-spatial and spatial reasoning, at the cost of significant formal overhead. In contrast, we avoid the overhead by keeping most of the reasoning about graphs at the non-spatial level.

Considering his first proof too complex, Yang tackled Schorr-Waite again using *relational* separation logic [Yang 2007]. Relational logic establishes a contextual refinement between two programs; in the case of Schorr-Waite, a refinement with the obvious depth-first-search (DFS) implementation of graph marking. While the resulting proof achieved a conceptual simplification over the original non-relational proof, this is somewhat counter-intuitive, as refinement between two programs is *generally* a much stronger property—and thus more demanding to prove—than establishing a pre- and postcondition for a program. Our result confirms this intuition. In comparison, an optimization of Yang's relational proof is given by Crespo and Kunz [2011], who show that Schorr-Waite is contextually equivalent to DFS, and mechanize the proof in around 3000 lines in Coq.

Non-separation proofs of Schorr-Waite. Being a standard verification benchmark for graph algorithms, Schorr-Waite has been verified numerous times, using a variety of different approaches. These ranged from automated ones [Leino 2010; Loginov et al. 2006; Roever 1977; Suzuki 1976], to studies of the algorithm's mathematical properties [Abrial 2003; Bornat 2000; Bubel 2007; Dershowitz 1980; Gries 1979; Griffiths 1979; Hubert and Marché 2005; Mehta and Nipkow 2003; Morris 1982; Topor 1979], to proofs based on program transformation [Broy and Pepper 1982; Dufourd 2014; Gerhart 1979; Giorgino et al. 2010; Preoteasa and Back 2012, 2010; Ward 1996]. The important difference from this work is that we explicitly wanted to support and utilize framing in our proof, as framing is the key feature of separation logic.

Proofs of union-find. The union-find structure [Galler and Fisher 1964] is also a well-studied graph benchmarks, with numerous correctness proofs, including in separation logic. For instance, Conchon and Filliâtre [2007] verify a persistent version in Coq that uses two functional arrays, while Lammich and Meis [2012] verify an implementation in Imperative/HOL. Krishnaswami [2011] provides a non-mechanized proof, and Charguéraud and Pottier [2019] give a Coq proof with a complexity analysis, both targeting a heap-allocated implementation, as we do. Wang et al. [2019], additionally verifies an array-based variant in Coq. Our approach differs in two key ways: we use partial graphs rather than traditional graph theory in order to leverage PCM morphisms; we also adopt small-footprint specifications that describe only the modified disjoint subsets, as opposed to the whole structure.

Graphs in separation logic, without PCMs. In the search for a *graph* predicate that enables decomposition, Bornat et al. [2004] define a partial graph as a recursive tree-shaped term, encoding the following strategy. Given some default traversal order, the first time a node is encountered, it's explicitly recorded in the term. Every later occurrence is recorded as a reference (i.e., pointer) to the first one. This enables that closed graphs can be encoded in a way that facilitates decomposition, but is dependent on the traversal order, which is problematic, as it prevents developing general libraries of lemmas about graphs. We used the PCM of partial graphs to remove the restriction to closed graphs, which also removes the traversal-order dependence.

Wang [2019] and Wang et al. [2019] parametrize their *graph* predicate with a closed mathematical graph, and express program invariant in terms of it. Because a closed graph doesn't necessarily decompose into closed subgraphs, these invariants aren't considered under distribution. Thus, to prove that a state modification maintains the graph invariant, one typically must resort to reasoning about the whole graph. This global reasoning can be ameliorated by introducing additional logical connectives and rules, such as e.g., overlapping conjunction and ramified frame rules [Hobor and Villard 2013]. In contrast, by relying on PCMs and morphisms, we were able to stay within standard separation logic over heaps.

Graphs in separation logic, with PCMs. Sergey et al. [2015a] and Nanevski [2016] parametrized their *graph* predicate by a heap (itself a PCM), that serves as a representation of a partial graph. Heaps obviously decompose under disjoint union. However, since the *graph* predicate includes the restriction that the implemented graph must be closed, the predicate itself doesn't distribute. This is worked around by introducing helper relations for when one graph is a subgraph of another, which our formulation provides for free (e.g., γ_1 is a subgraph of γ iff $\gamma = \gamma_1 \bullet \gamma_2$ for some γ_2). These works don't consider PCM morphisms or Schorr-Waite.

More recently, Krishna et al. [2020] and Meyer et al. [2023] introduced a theory of *flows*, as a framework to study the decomposition of graph properties. Flows evoke Yang's first proof, in that they serve as decomposition-supporting proxies for the specification of the graph. In contrast, we parametrize the reasoning by the whole graph, and compute proxies by morphisms, when needed. That said, flows are intended for automated reasoning, which we haven't considered.

Finally, Costa et al. [[n. d.]] define the PCM of *pregraphs*. A pregraph is like our partial graph, but it may contain incoming dangling edges, not just outgoing ones that sufficed for us. The paper proceeds to define a separation logic over pregraphs, much as separation logic over heaps is classically defined using local actions [Calcagno et al. 2007]. In contrast, we use a standard separation logic over heaps, where (partial) graphs are merely a secondary user-level PCM. Having heaps and graphs coexist is necessary for representing pointer-based graph algorithms such as Schorr-Waite, but it requires PCM morphisms to mediate between the two.

Morphisms in separation logic. While extant separation logics extensively rely on PCMs, PCM morphisms remains underutilized. A notable exception is the work of Farka et al. [2021], who develop a theory of *partial PCM morphisms*—morphisms that distribute only under certain conditions. These conditions give rise to *separating relations*, a novel algebraic structure with rich theoretical properties. Together, morphisms and separating relations define when one PCM is a sub-PCM of another, and how to refine Hoare logic triples accordingly.

This refinement concept originated with Nanevski et al. [2019], who formulated it as a morphism over *resources*, which are algebraic structures modeling state-transition systems for concurrency.

Although resources involve a form of structural inclusion, they don't form PCMs, nor does their theory rely on join-based decomposition. Moreover, resource morphisms act on programs, to retroactively adapt a program's ghost code via a simulation function, whereas PCM morphisms operate within logical assertions. As a result, resources and their morphisms address orthogonal concerns to ours. Finally, neither of the works applies their morphisms to graphs.

Graphs algebraically. Recent works in functional programming, theorem proving and category theory have proposed treating graphs algebraically [Liell-Cock and Schrijvers 2024; Master 2021, 2022; Mokhov 2017, 2022], though so far without application to separation logic. Most recently, Kidney and Wu [2025] share with us the representation of graphs as maps, which they further endow with the algebraic structure of rings, comprising distinct monoids for vertices and for edges. They also propose several different algebraic constructions along with the associated morphisms. Their application is in implementing graph algorithms as state-free coinductive programs in Agda and Haskell. In contrast, we use the algebra of graphs for specification and verification of programs, not for their implementation, as our programs in general, and our variant of Schorr-Waite in particular, are implemented in an imperative pointer-based language customary for separation logic. Our focus on separation logic also gives rise to somewhat different monoidal structures. In particular, as we need to relate graphs to pointers and framing, we focus on partiality and, correspondingly, use a monoid of graphs whose join operation is very different from the constructions considered in the above work.

9 Conclusion

This paper establishes that graphs form a PCM when extended with *dangling edges*, yielding *partial graphs*. The PCM structure facilitates natural composition through subgraph joins, and induces *PCM morphisms* as structure-preserving functions. Our central contribution demonstrates how PCM morphisms address separation logic's long-standing challenge of effective graph verification.

Crucially, PCM morphisms enrich the foundational principle of locality: while traditional framing enables spatial locality by isolating heap portions, morphisms enable non-spatial locality by isolating relevant graph subcomponents. The key mechanism is *contextual localization*; that is, distributing morphisms across subgraph joins ($f(\gamma_1 \bullet \gamma_2) = f\gamma_1 \bullet f\gamma_2$). Unlike framing, which operates only at the top level of a specification, contextual localization supports rewriting deeply inside a context.

We further employ *higher-order combinators* like *map* to define helper notions for complex graph invariants (e.g., in Schorr-Waite) and to support general lemmas that can be reused across multiple instances. The integration of partial graphs, morphisms, and higher-order combinators yields *novel, mechanized proofs* for both *Schorr-Waite algorithm* and the *union-find data structure*, achieving substantial conciseness through lemma reuse and contextual localization.

All proofs are fully mechanized in Coq using the Hoare Type Theory library. The mechanization uses functional (non-mutable) variables in place of mutable ones used in the paper, but this is a technical variation with no impact on the results.

Acknowledgments

We thank Anindya Banerjee and the anonymous ICFP reviewers for their numerous useful comments. This work was partially supported by the Spanish MICIU grants DECO PID2022-138072OB-I00, Maria de Maeztu CEX2024-001471-M, PRODIGY TED2021-132464B-I00, and the ERC Starting Grant CRETE GA:101039196. Any views, opinions and findings expressed in the text are those of the authors and don't necessarily reflect the views of the funding agencies.

References

Jean-Raymond Abrial. 2003. Event based sequential program development: application to constructing a pointer program. In *Formal Methods Europe (FME)*. 51–74. doi:10.1007/978-3-540-45236-2_5

Richard Bornat. 2000. Proving pointer programs in Hoare logic. In *Mathematics of Program Construction (MPC)*. 102–126. doi:10.1007/10722010_8

Richard Bornat, Cristiano Calcagno, and Peter O'Hearn. 2004. Local reasoning, separation and aliasing. In *Workshop on Semantics, Program Analysis and Computing Environments for Memory Management (SPACE)*. http://www.cs.ucl.ac.uk/staff/p.ohearn/papers/separation-and-aliasing.pdf

Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. 2005. Permission accounting in separation logic. In *Principles of Programming Languages (POPL)*. 259–270. doi:10.1145/1040305.1040327

Manfred Broy and Peter Pepper. 1982. Combining algebraic and algorithmic reasoning: an approach to the Schorr-Waite algorithm. *ACM Trans. Program. Lang. Syst.* 4, 3 (1982), 362–381. doi:10.1145/357172.357175

Richard Bubel. 2007. The Schorr-Waite-Algorithm. In *Verification of Object-Oriented Software: The KeY Approach*. 569–587. doi:10.1007/978-3-540-69061-0_15

Cristiano Calcagno, Peter W. O'Hearn, and Hongseok Yang. 2007. Local action and abstract separation logic. In *Logic in Computer Science (LICS)*. 366–378. doi:10.1109/LICS.2007.30

Arthur Charguéraud and François Pottier. 2019. Verifying the correctness and amortized complexity of a union-find implementation in separation logic with time credits. *J. Autom. Reasoning* 62 (2019), 331–365. doi:10.1007/s10817-017-9431-7

Sylvain Conchon and Jean-Christophe Filliâtre. 2007. A persistent union-find data structure. In *ML Workshop*. 37–46. doi:10.1145/1292535.1292541

Diana Costa, James Brotherston, and David Pym. [n. d.]. *Graph decomposition and local reasoning*. Retrieved 2024 from http://www0.cs.ucl.ac.uk/staff/D.Pym/pregraphs.pdf

Juan Manuel Crespo and César Kunz. 2011. A machine-checked framework for relational separation logic. In *Software Engineering and Formal Methods (SEFM)*. 122–137. doi:10.1007/978-3-642-24690-6_10

Nachum Dershowitz. 1980. The Schorr-Waite marking algorithm revisited. *Inform. Process. Lett.* 11, 3 (1980), 141–143. doi:10.1016/0020-0190(80)90130-1

Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew Parkinson, and Hongseok Yang. 2013. Views: compositional reasoning for concurrent programs. In *Principles of Programming Languages (POPL)*. 287–300. doi:10.1145/2429069.2429104

Jean-François Dufourd. 2014. Pointer program derivation using Coq: graphs and Schorr-Waite algorithm. In *Formal Methods and Software Engineering (ICFEM)*. 139–154. doi:10.1007/978-3-319-11737-9_10

František Farka, Aleksandar Nanevski, Anindya Banerjee, Germán Andrés Delbianco, and Ignacio Fábregas. 2021. On algebraic abstractions for concurrent separation logics. *Proc. ACM Program. Lang.* 5, POPL, Article 5 (2021). doi:10.1145/3434286

Bernard A. Galler and Michael J. Fisher. 1964. An improved equivalence algorithm. *Commun. ACM* 7, 5 (1964), 301–303. doi:10.1145/364099.364331

Susan L. Gerhart. 1979. A derivation-oriented proof of the Schorr-Waite marking algorithm. In *Program Construction: International Summer School*. 472–492. doi:10.1007/BFb0014678

Mathieu Giorgino, Martin Strecker, Ralph Matthes, and Marc Pantel. 2010. Verification of the Schorr-Waite algorithm - from trees to graphs. In *Logic-Based Program Synthesis and Transformation (LOPSTR)*. 67–83. doi:10.1007/978-3-642-20551-4_5

Marcos Grandury, Aleksandar Nanevski, and Alexander Gryzlov. 2025a. *Artifact for Verifying graph algorithms in separation logic: a case for an algebraic approach*. doi:10.5281/zenodo.15847649

Marcos Grandury, Aleksandar Nanevski, and Alexander Gryzlov. 2025b. Verifying graph algorithms in separation logic: a case for an algebraic approach (extended version). (2025). arXiv:2501.13603 [cs.LO] https://arxiv.org/abs/2501.13603

David Gries. 1979. The Schorr-Waite graph marking algorithm. In *Program Construction: International Summer School*. 58–69. doi:10.1007/BFb0014658

Michael Griffiths. 1979. Development of the Schorr-Waite algorithm. In *Program Construction: International Summer School*. 464–471. doi:10.1007/BFb0014677

Aquinas Hobor and Jules Villard. 2013. The ramifications of sharing in data structures. In *Principles of Programming Languages (POPL)*. 523–536. doi:10.1145/2429069.2429131

HTT 2010. Hoare Type Theory. Retrieved 2025 from https://github.com/imdea-software/htt

Thierry Hubert and Claude Marché. 2005. A case study of C source code verification: the Schorr-Waite algorithm. *Software Engineering and Formal Methods (SEFM)* (2005), 190–199. doi:10.1109/SEFM.2005.1

Samin Ishtiaq and Peter W. O'Hearn. 2001. BI as an assertion language for mutable data structures. In *Principles of Programming Languages (POPL)*. 14–26. doi:10.1145/360204.375719

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: monoids and invariants as an orthogonal basis for concurrent reasoning. In *Principles of Porgramming Languages (POPL)*. 637–650. doi:10.1145/2676726.2676980

Donnacha Oisín Kidney and Nicolas Wu. 2025. Formalising graph algorithms with coinduction. *Proc. ACM Program. Lang.* 9, POPL, Article 56 (2025). doi:10.1145/3704892

Siddharth Krishna, Alexander J. Summers, and Thomas Wies. 2020. Local reasoning for global graph properties. In *European Symposium on Programming (ESOP)*. 308–335. doi:10.1007/978-3-030-44914-8_12

Neelakantan R. Krishnaswami. 2011. *Verifying higher-order imperative programs with higher-order separation logic.* Ph. D. Dissertation. Carnegie Mellon University. http://reports-archive.adm.cs.cmu.edu/anon/2012/CMU-CS-12-127.pdf

Peter Lammich and Rene Meis. 2012. A separation logic framework for Imperative HOL. *Arch. Formal Proofs* (2012). https://www.isa-afp.org/entries/Separation_Logic_Imperative_HOL.shtml

K. Rustan M. Leino. 2010. Dafny: an automatic program verifier for functional correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*. 348–370. doi:10.1007/978-3-642-17511-4_20

Jack Liell-Cock and Tom Schrijvers. 2024. Let a thousand flowers bloom: an algebraic representation for edge graphs. *The Art, Science, and Engineering of Programming* 8, 3 (2024). doi:10.22152/programming-journal.org/2024/8/9

Alexey Loginov, Thomas W. Reps, and Mooly Sagiv. 2006. Automated verification of the Deutsch-Schorr-Waite tree-traversal algorithm. In *Static Analysis Symposium (SAS)*. 261–279. doi:10.1007/11823230_17

Jade Master. 2021. The open algebraic path problem. doi:10.48550/arXiv.2005.06682

Jade Master. 2022. How to compose shortest paths. doi:10.48550/arXiv.2205.15306

Farhad Mehta and Tobias Nipkow. 2003. Proving pointer programs in higher-order logic. In *Conference on Automated Deduction (CADE)*. 121–135. doi:10.1007/978-3-540-45085-6_10

Roland Meyer, Thomas Wies, and Sebastian Wolff. 2023. Make flows small again: revisiting the flow framework. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. 628–646. doi:10.1007/978-3-031-30823-9_32

Andrey Mokhov. 2017. Algebraic graphs with class (functional pearl). In *International Symposium on Haskell.* 2–13. doi:10.1145/3122955.3122956

Andrey Mokhov. 2022. United monoids: finding simplicial sets and labelled algebraic graphs in trees. *The Art, Science, and Engineering of Programming* 6, 3 (2022). doi:10.22152/programming-journal.org/2022/6/12

Joseph M. Morris. 1982. A proof of the Schorr-Waite algorithm. In *Theoretical Foundations of Programming Methodology: Lecture Notes of an International Summer School.* 43–51. doi:10.1007/978-94-009-7893-5_5

Aleksandar Nanevski. 2016. Separation logic and concurrency. Lecture notes for the Oregon Programming Languages Summer School (OPLSS). https://www.cs.uoregon.edu/research/summerschool/summer16/curriculum.php

Aleksandar Nanevski, Anindya Banerjee, Germán Andrés Delbianco, and Ignacio Fábregas. 2019. Specifying concurrent programs in separation logic: morphisms and simulations. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 161 (2019). doi:10.1145/3360587

Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. 2006. Polymorphism and separation in Hoare Type Theory. In *International Conference on Functional Programming (ICFP)*. 62–73. doi:10.1145/1159803.1159812

Aleksandar Nanevski, Greg Morrisett, Avraham Shinnar, Paul Govereau, and Lars Birkedal. 2008. Ynot: dependent types for imperative programs. In *International Conference on Functional Programming (ICFP)*. 229–240. doi:10.1145/1411203.1411237

Aleksandar Nanevski, Viktor Vafeiadis, and Josh Berdine. 2010. Structuring the verification of heap-manipulating programs. In *Principles of Programming Languages (POPL)*. 261–274. doi:10.1145/1706299.1706331

Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. 2001. Local reasoning about programs that alter data structures. In *Computer Science Logic (CSL)*. 1–19. doi:10.1007/3-540-44802-0_1

Viorel Preoteasa and Ralph-Johan Back. 2012. Invariant diagrams with data refinement. *Formal Aspects Comput.* 24, 1 (2012), 67–95. doi:10.1007/S00165-011-0195-2

Viorel Preoteasa and Ralph-Johan Back. 2010. Verification of the Deutsch-Schorr-Waite graph marking algorithm using data refinement. *Arch. Formal Proofs* (2010). https://www.isa-afp.org/browser_info/current/AFP/GraphMarkingIBP/document.pdf

John C. Reynolds. 2002. Separation logic: a logic for shared mutable data structures. In *Logic in Computer Science (LICS)*. 55–74. doi:10.1109/LICS.2002.1029817

Willem-Paul Roever. 1977. On backtracking and greatest fixpoints. In *International Colloquium on Automata, Languages and Programming (ICALP)*. 412–429. doi:10.1007/3-540-08342-1_32

Herbert Schorr and William M. Waite. 1967. An efficient machine-independent procedure for garbage collection in various list structures. *Commun. ACM* 10, 8 (1967), 501–506. doi:10.1145/363534.363554

Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. 2015a. Mechanized verification of fine-grained concurrent programs. In *Programming Language Design and Implementation (PLDI)*. 77–87. doi:10.1145/2737924.2737964

Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. 2015b. Specifying and verifying concurrent algorithms with histories and subjectivity. In *European Symposium on Programming (ESOP)*. 333–358. doi:10.1007/978-3-662-46669-8_14

Norihisa Suzuki. 1976. *Automatic verification of programs with complex data structures.* Ph. D. Dissertation. Stanford University.

Rodney W. Topor. 1979. The correctness of the Schorr-Waite list marking algorithm. *Acta Informatica* 11 (1979), 211–221. doi:10.1007/BF00289067

Shengyi Wang. 2019. *Mechanized verification of graph-manipulating programs.* Ph. D. Dissertation. School of Computing, National University of Singapore. https://scholarbank.nus.edu.sg/handle/10635/166280

Shengyi Wang, Qinxiang Cao, Anshuman Mohan, and Aquinas Hobor. 2019. Certifying graph-manipulating C programs via localizations within data structures. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 171 (2019). doi:10.1145/3360597

Martin Ward. 1996. Derivation of data intensive algorithms by formal transformation: the Schorr-Waite graph marking algorithm. *IEEE Trans. Softw. Eng.* 22, 9 (1996), 665–686. doi:10.1109/32.541437

Hongseok Yang. 2001a. An example of local reasoning in BI pointer logic: the Schorr-Waite graph marking algorithm. In *Workshop on Semantics, Program Analysis and Computing Environments for Memory Management (SPACE).* https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=d93456b70d4ec3569c29409c73aabb0fddb6ed49

Hongseok Yang. 2001b. *Local reasoning for stateful programs.* Ph. D. Dissertation. University of Illinois at Urbana-Champaign. https://drive.google.com/open?id=1gFdENbez-p6tCksvz2NgdU7y_aV0zHLW

Hongseok Yang. 2007. Relational separation logic. *Theor. Comput. Sci.* 375 (2007), 308–334. doi:10.1016/j.tcs.2006.12.036

Received 2025-02-21; accepted 2025-06-27