Certifying Graph-Manipulating C Programs via Localizations within Data Structures

SHENGYI WANG, National University of Singapore, Singapore
QINXIANG CAO, Shanghai Jiao Tong University, China
ANSHUMAN MOHAN, National University of Singapore, Singapore
AQUINAS HOBOR, National University of Singapore, Singapore

We develop powerful and general techniques to mechanically verify realistic programs that manipulate heap-represented graphs. These graphs can exhibit well-known organization principles, such as being a directed acyclic graph or a disjoint-forest; alternatively, these graphs can be totally unstructured. The common thread for such structures is that they exhibit deep intrinsic sharing and can be expressed using the language of graph theory. We construct a modular and general setup for reasoning about abstract mathematical graphs and use separation logic to define how such abstract graphs are represented concretely in the heap. We develop a Localize rule that enables modular reasoning about such programs, and show how this rule can support existential quantifiers in postconditions and smoothly handle modified program variables. We demonstrate the generality and power of our techniques by integrating them into the Verified Software Toolchain and certifying the correctness of seven graph-manipulating programs written in CompCert C, including a 400-line generational garbage collector for the CertiCoq project. While doing so, we identify two places where the semantics of C is too weak to define generational garbage collectors of the sort used in the OCaml runtime. Our proofs are entirely machine-checked in Coq.

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

Additional Key Words and Phrases: Separation logic, Graph-manipulating programs, Coq, CompCert, VST

ACM Reference Format:
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 (October 2019), 30 pages. https://doi.org/10.1145/3360597

1 INTRODUCTION

Over the last fifteen years, separation logic has facilitated great strides in verifying programs that manipulate tree-shaped data structures. [Appel et al. 2014; Bengtson et al. 2012; Berdine et al. 2005; Chin et al. 2010; Chlipala 2011; Jacobs et al. 2011]. Unfortunately, programs that manipulate graph-shaped data structures (*i.e.* structures with *intrinsic sharing*) have proved harder to verify. Indeed, such programs were formidable enough that many of the early landmark results in separation logic devoted substantial effort to verifying single examples such as Schorr-Waite [Yang 2001] or union-find [Krishnaswami 2011] with pen and paper. More recent landmarks have moved to a machine-checked context, but have still been devoted to either single examples or to classes of

Authors' addresses: Shengyi Wang, National University of Singapore, Singapore; Qinxiang Cao, Shanghai Jiao Tong University, China; Anshuman Mohan, National University of Singapore, Singapore; Aquinas Hobor, National University of Singapore, Singapore.

2475-1421/2019/10-ART171
https://doi.org/10.1145/3360597

closely-related examples such as garbage collectors [Ericsson et al. 2017; McCreight et al. 2010]. These kinds of examples tend to require a large number of custom predicates and subtle reasoning, which generally does not carry over to the verification of other graph-manipulating programs.

In contrast, we present a toolkit for verifying graph-manipulating programs in a machine-checked context. Our techniques are *general* in that they handle a diverse range of programs, and *modular* in that they encourage code reuse (*e.g.* facts about reachability) and separation of concerns (*e.g.* between abstract mathematical graphs and concrete representations in the heap). Our techniques are *powerful* enough to reason about real C code as compiled by CompCert [Leroy 2006], and also *lightweight* enough to integrate into the Verified Software Toolchain (VST) [Appel et al. 2014] without major reengineering. CompCert and VST are distributed via opam and the CoqIDE installer, so they have a sizable userbase that can take advantage of our techniques. Finally, our techniques *scale* well beyond short toy programs: we certify the correctness of a generational garbage collector for the CertiCoq project [Anand et al. 2017] (≈ 400 rather devilish lines of C).

We proceed in three steps. First, we develop a mathematical graph library that is general enough to reason about a wide variety of algorithms and expressive enough to describe the behavior of these algorithms in real machines. We modularize this library carefully so that common ideas—*e.g.* subgraphs, reachability, and isomorphism—can be efficiently reused in different algorithms. Second, we use separation logic to express how these abstract graphs are actualized in the heap as concrete graphs in a way that facilitates the reuse of key definitions and theorems across algorithms. Third, we develop a notion of *localization blocks* that enables modular reasoning in our Hoare proofs, even in the presence of the implicit sharing intrinsic to graphs, by using our Localize rule:

$$\textsc{Localize}\quad \frac{G_1 \vdash L_1 * R \qquad \{L_1\}\ c\ \{\exists x.\ L_2\} \qquad R \vdash \forall x.\ (L_2 \mathrel{-\!\!*} G_2)}{\{G_1\}\ c\ \{\exists x.\ G_2\}}\ \mathsf{FreeVar}(R) \cap \mathsf{ModVar}(c) = \varnothing \tag{1}$$

Localize connects the "local" effect of a command c, *i.e.* transforming L_1 to L_2, with its "global" effect, *i.e.* from G_1 to G_2. Localize is a more general version of the well-known Frame rule, which does the same task in the simpler case when $G_i = L_i * F$ for some frame F that is untouched by c. Localize can handle the more general transformation if we can find a *ramification frame* R that satisfies a pair of delicately-stated entailments[1] and the side condition on modified local program variables. Localize upgrades the Ramify rule [Hobor and Villard 2013] in two key ways: support for existential quantifiers in postconditions and smoother treatment of modified program variables.

Our contributions are organized as follows:

- §2 We use the classic "union-find" disjoint set algorithm to show how our three key ingredients—mathematical graphs, spatial graphs, and localization blocks—come together to verify graph-manipulating algorithms. To the best of our knowledge this is the first machine-checked verification of this algorithm that starts with real C code. We briefly review the seven programs we have verified to give a sense of the breadth of algorithms our system can tackle.
- §3 We show that Localize and Frame are co-derivable. We illustrate a delicate technique to properly handle modified local variables. We show a mark-graph program that explores a graph in a fold/unfold style, and discuss the utility of linked existentials in postconditions.
- §4 We develop a general and modular framework of mathematical graphs powerful enough to support realistic verification in a mechanized context. We give a sampling of key definitions.
- §5 We suggest that the Knaster-Tarski fixpoint [Tarski 1955] cannot define a usable separation logic graph predicate. We propose a better definition for general spatial graphs that still

[1] Readers less familiar with the separating implication $P \mathrel{-\!\!*} Q$, also known as *magic wand*, can refer to its semantics in Figure 8 (page 13), which also models the other separation logic operators we use in this paper. The key proof rule for magic wand is its adjointness with the separating conjunction: $(P * Q \vdash R) \Leftrightarrow (P \vdash Q \mathrel{-\!\!*} R)$.

```
1   struct Node { unsigned int rank;
2                 struct Node * parent; }
3   // $\{\mathrm{uf\_graph}(\gamma) \land \mathtt{x} \in V(\gamma)\}$
4   struct Node* find(struct Node* x) {
5     struct Node *p;
6   // $\{\mathrm{uf\_graph}(\gamma) \land \mathtt{x} \in V(\gamma) \land \exists r, pa.\ \gamma(\mathtt{x}) = (r, pa) \land pa \in V(\gamma)\}$
7   // $\searrow$ $\{\mathtt{x} \mapsto r, pa \land \mathtt{x} \in V(\gamma) \land \gamma(\mathtt{x}) = (r, pa) \land pa \in V(\gamma)\}$
8   $\lightning$(2)  p = x -> parent;
9   // $\swarrow$ $\{\mathtt{x} \mapsto r, pa \land \mathtt{p} = pa \land \mathtt{x} \in V(\gamma) \land \gamma(\mathtt{x}) = (r, pa) \land pa \in V(\gamma)\}$
10  // $\{\mathrm{uf\_graph}(\gamma) \land \mathtt{p} = pa \land \mathtt{x} \in V(\gamma) \land \gamma(\mathtt{x}) = (r, pa) \land pa \in V(\gamma)\}$
11    if (p != x) {
12  // $\{\mathrm{uf\_graph}(\gamma) \land \mathtt{p} = pa \land pa \neq \mathtt{x} \land \mathtt{x} \in V(\gamma) \land \gamma(\mathtt{x}) = (r, pa) \land pa \in V(\gamma)\}$
13      p = find(p);
14  // $\{\exists \gamma', rt.\ \mathrm{uf\_graph}(\gamma') \land \mathtt{p} = rt \land pa \neq \mathtt{x} \land \mathtt{x} \in V(\gamma) \land \mathit{findS}(\gamma, pa, \gamma') \land \mathit{uf\_root}(\gamma', pa, rt) \land \gamma(\mathtt{x}) = (r, pa)\}$
15  // $\searrow$ $\{\mathtt{x} \mapsto r, pa \land \mathtt{p} = rt \land pa \neq \mathtt{x} \land \mathit{findS}(\gamma, pa, \gamma') \land \mathit{uf\_root}(\gamma', pa, rt) \land \mathtt{x} \in V(\gamma) \land \gamma(\mathtt{x}) = (r, pa)\}$
16  $\lightning$(3)  x -> parent = p;
17  // $\swarrow$ $\{\mathtt{x} \mapsto r, rt \land \mathtt{p} = rt \land pa \neq \mathtt{x} \land \mathit{findS}(\gamma, pa, \gamma') \land \mathit{uf\_root}(\gamma', pa, rt) \land \mathtt{x} \in V(\gamma) \land \gamma(\mathtt{x}) = (r, pa)\}$
18  // $\{\exists \gamma''.\ \mathrm{uf\_graph}(\gamma'') \land \mathit{findS}(\gamma, pa, \gamma'') \land \mathit{uf\_root}(\gamma'', \mathtt{x}, rt) \land \mathtt{p} = rt\}$
19    } return p;
20  }  // $\{\exists \gamma'', rt.\ \mathrm{uf\_graph}(\gamma'') \land \mathit{findS}(\gamma, \mathtt{x}, \gamma'') \land \mathit{uf\_root}(\gamma'', \mathtt{x}, rt) \land \mathtt{ret} = rt\}$
```

$$\mathrm{uf\_graph}(x, \gamma) \triangleq \mathop{\Huge *}_{v \in V(\gamma)} v \mapsto \gamma(v)$$

$$\mathit{uf\_root}(\gamma, x, rt) \triangleq x \overset{\gamma}{\leadsto}{}^{\star} rt \ \land \ \forall rt'.\ rt \overset{\gamma}{\leadsto}{}^{\star} rt' \Rightarrow rt = rt'$$

$$\begin{aligned} \mathit{findS}(\gamma, x, \gamma') \triangleq\ & (\forall v.\ v \in V(\gamma) \Leftrightarrow v \in V(\gamma')) \land \\ & (\forall v.\ v \in V(\gamma) \Rightarrow \gamma(v).\mathit{rank} = \gamma'(v).\mathit{rank}) \land \\ & (\forall r, r'.\ \mathit{uf\_root}(\gamma, v, r) \Rightarrow \mathit{uf\_root}(\gamma', v, r') \Rightarrow r = r') \land \\ & (\gamma \setminus \{v \in \gamma \mid x \overset{\gamma}{\leadsto}{}^{\star} v\} \cong \gamma' \setminus \{v \in \gamma \mid x \overset{\gamma}{\leadsto}{}^{\star} v\}) \end{aligned}$$

Fig. 1. Clight code and proof sketch for `find`; red text indicates the line-by-line changes

enjoys a "recursive" fold/unfold. We prove general theorems about spatial graphs that are organized in a modular way that can be utilized in multiple flavors of separation logic.

§6 We discuss our flagship example, the certification of the CertiCoq garbage collector (GC). We identify two places where the semantics of C is too weak to define an OCaml-style GC. We also find and fix a rather subtle overflow error in the original C code for the GC, thereby justifying the effort of developing machine-checked proofs of correctness.

§7 We discuss how our techniques are integrated into the "Floyd" module of VST, a separation-logic based engine to help users verify CompCert C programs, via two new Floyd tactics `localize` and `unlocalize`. We also document statistics related to our overall development.

§8 We discuss related work.

§9 We discuss directions for future work and conclude.

All of our results are machine checked in Coq and are available as an artifact online. An extended version of this paper featuring three appendices is also available online [Wang et al. 2019].

2 TOUR OF A VERIFIED EXAMPLE

Before jumping into an involved discussion of our three-part recipe, we first (§2.1) build intuition by showing how they are applied to verify the union-find algorithm. We then (§2.2) explain our new *localization blocks*. Finally (§2.3), we briefly discuss the other examples we have verified.

2.1 Localizations Yield a Tidy Union-Find

As an initial demonstration of our techniques, we show the decorated code of the `find` function from the classic disjoint-set data structure [Cormen et al. 2009] in Figure 1. The function returns the root (ultimate parent) of a `Node x`. A node is a root whose parent pointer points to itself (line 11). Other than such self-loops at roots, the structure is acyclic. For good amortised performance, `find` performs path compression (line 16). While `find` may appear straightforward—the code is short, and a `Node` only has one outgoing pointer—the disjoint-set data structure is tricky to reason about because of the subtle nature of path compression and the implicit sharing inherent in parent-pointers. Indeed, the first pen-and-paper verification in separation logic required 20 pages [Krishnaswami 2011].

We use the following conventions in our invariants. Pure predicates are written in *italic*. We write γ to mean a "mathematical" (or "pure") graph: roughly, a set of labeled vertices $V(\gamma)$ and edges $E(\gamma)$. When $v \in V(\gamma)$, we write $\gamma(v) = (r, pa)$ to state that vertex v has label r and parent vertex pa (r stores the "rank" of a node; it is ignored in `find`). We detail mathematical graphs in §4.

Spatial predicates are written in sans-serif. Each node $v \in V(\gamma)$ is represented in the heap by $v \mapsto \gamma(v)$, where we employ the usual pen-and-paper shorthand of writing *e.g.* $v \mapsto r, pa$ to mean $\big(v \mapsto r\big) * \big((v + \mathit{sizeof}(\texttt{unsigned int})) \mapsto pa\big)$ in the character-addressed C memory model. The whole graph (disjoint-set forest) is represented by $\mathsf{uf\_graph}(\gamma)$, essentially the iterated separating conjunction of the representations of each vertex $v \in V(\gamma)$. We detail spatial graphs in §5.

The invariants at each program point are natural despite only minor tidying from our machine-checked proof. We also enjoy good separation between the spatial predicates and pure predicates. All of this is despite verifying real C code, which entails quite a number of grungy details. As one example, we will shortly examine some grunginess that occurs in the verification of line 11.

The precondition, stated on line 3, says that we have a disjoint-set forest representing the abstract graph γ, and that x is a valid vertex in γ. The postcondition is on line 20: the heap contains a new union-find graph γ'', and `find` returns the node rt. We specify that rt is the root (ultimate parent) of x with the mathematical relation *uf_root*. The mathematical relation *findS*, which conservatively approximates the action of path compression, relates the final graph γ'' to the original graph γ. The formal definitions for the concepts used in *uf_root* and *findS* will be given in §4, but briefly: $x \overset{\gamma}{\leadsto}^{\star} y$ expresses that y is reachable from x in γ, $\gamma \setminus S$ expresses the result of removing the vertices in set S from graph γ, and $\gamma_1 \cong \gamma_2$ expresses that the two graphs are structurally equivalent.

Most of the verification is straightforward. For readability, we mark line-by-line changes in the invariants in red. Each line of code (8, 11, 13, 16, and 19) is bracketed with invariants leading to relatively easy proofs of the command (ignoring the symbols $\searrow$, $\lightning(i)$, and $\swarrow$ until §2.2). In addition to improving human comprehensibility, this also aids mechanical comprehensibility: straightforward invariants help the underlying verification engine (VST, in our case) handle many grungy details for us either automatically or with a little human guidance via suitable lemmas.

The pointer comparison in line 11 is an example of where such lemmas are necessary. Formally, pointer (in-)equality comparison in C is only defined under somewhat delicate circumstances[2]. VST could prove the definedness of the pointer comparison automatically if we knew $(\mathsf{x} \mapsto \_) * (\mathsf{p} \mapsto \_) * \top$, but unfortunately this does not follow from line 9 since, when x is a root, self-loop gives us $\mathsf{x} = \mathsf{p}$ and $\mathsf{x} \mapsto \_ * \mathsf{x} \mapsto \_ \vdash \bot$. Accordingly, we must prove a simple lemma that states that when $\mathsf{uf\_graph}(\gamma) \wedge \mathsf{x} \in V(\gamma) \wedge \mathsf{p} \in V(\gamma)$, the pointer comparison is defined in C.

2.2 Localization Blocks

It is time to explain the non-obvious jumps in reasoning bracketed by the symbols $\searrow$, $\lightning(i)$, and $\swarrow$ (lines 6–10 and 14–18). We call such bracketed sets of lines "localization blocks", and the $\searrow$ and $\swarrow$ symbols formally indicate an application of the Localize rule (Equation 1 from page 2). As explained above, the verification of the command itself is entirely straightforward given its immediate neighbors (lines 7–9 and 15–17). What is not so straightforward is how *e.g.* line 6 leads to line 7 or how line 9 leads to line 10. To give intuition, a localization block allows us to zoom in from a larger

[2] Executive summary: it is a mess. Full story: whenever (1) x and p are both null; or when (2) one of them is null and the other has offset between 0 and the size of the memory block into which it is pointing; or when (3) if x and p are from the same memory block, then both of their offsets are between 0 and the size of that block; or when (4) x and p are not in the same memory block and both have offsets between 0 and the size of their respective memory blocks *minus one.*

"global" context to a smaller "local" one, and, after verifying some commands locally and arriving at a local postcondition, to zoom back out to the global context.

Recall that Localize connects some "global" pre- and postconditions G_1 and G_2 with some "local" pre- and postconditions L_1 and L_2 using a ramification frame R. The lines adjacent to the $\searrow$ and $\swarrow$ symbols specify G_1 (*e.g.* line 6), L_1 (line 7), L_2 (line 9), and G_2 (line 10). Note that the Localize rule expects quantifiers in the postconditions L_2 and G_2, but lines 9–10 do not have any. We can overcome this mismatch since $\forall P.\ (P \dashv\vdash \exists x : unit.P)$ for any x not free in P.

We must now pick a ramification frame R that satisfies the entailments $G_1 \vdash L_1 * R$ and (again eliding the quantifier) $R \vdash L_2 \mathrel{-\!\!*} G_2$. This is delicate. To give intuition, it is **almost** enough to choose $R \triangleq L_2 \mathrel{-\!\!*} G_2$, which makes the second entailment trivial, and leaves us with three checks.

The first check is the *ramification entailment*, $G_1 \vdash L_1 * (L_2 \mathrel{-\!\!*} G_2)$, which asks whether replacing L_1 with L_2 inside G_1 yields G_2. This is a nontrivial proof obligation both because we must prove that L_1 is located inside G_1, and that "replacing" L_1 with L_2 yields G_2. Henceforth, we refer to the ramification entailment of a localization block by using the symbol $\lightning(i)$ to connect it to a relevant equation. Accordingly, $\lightning(2)$ refers to the ramification entailment associated with lines 6–10:

$$\mathsf{graph}(\gamma) \wedge x \in V(\gamma) \quad \vdash \quad x \mapsto \gamma(x) * \big(x \mapsto \gamma(x) \mathrel{-\!\!*} \mathsf{graph}(\gamma)\big) \tag{2}$$

Here we have isolated the key **spatial** parts of the invariants on lines 6–10. Notice that this lemma is stated for any whole-graph predicate $\mathsf{graph}(\gamma)$, and not merely for the special class of "union-find graphs" $\mathsf{uf\_graph}(\gamma)$ (that *e.g.* have only one outgoing edge per node). That is useful because we use the same lemma to prove similar goals in all of our examples. Indeed, this "unchanged vertex" ramification entailment is used whenever we need to read from a vertex in a graph. In §5 we describe other generic and reusable lemmas that prove other ramification entailments.

The second check is the Hoare proof of the local change from L_1 to L_2. Since lines 7 and 9 are straightforward—indeed, the point of Localize is to make them so—verifying line 8 is easy.

The third check is the side condition on the modified variables. Here we have an irritating problem: the free variables of $R \triangleq L_2 \mathrel{-\!\!*} G_2$ are **not** disjoint from the local variables modified by c. Inspection of lines 8–10 shows that the program variable p **is** modified by c, and **is** free in both L_2 and G_2. This issue is fundamental: the whole point of verifying a read is to know something about the value that has been read. Accordingly, our proof fails when we choose $R \triangleq (L_2 \mathrel{-\!\!*} G_2)$. We will address this problem head-on in §3.2, but for now let us content ourselves with knowing that other than this problem, Localize lets us verify lines 6–10.

The second localization block (lines 14–18) is both easier and harder than the first. It is easier because line 16 does not modify any local program variables, so the side condition is trivially satisfied. Moreover, although line 18 contains an existential, line 17 does not, and so there is no need to "link" the two associated witnesses. We will discuss this issue in more detail in §3.3, but for now it is enough to choose $R \triangleq L_2 \mathrel{-\!\!*} G_2$ exactly as written in lines 17 (for L_2) and 18 (for G_2).

On the other hand, the second localization block is harder than the first because there is more going on spatially. $\lightning(3)$ expresses an update to a single node of our graph:

$$\frac{x \in V(\gamma') \qquad \gamma'' = [x \to (r, rt)]\gamma'}{\mathsf{graph}(\gamma') \ \vdash\ x \mapsto \gamma'(x) * \big(x \mapsto \gamma''(x) \mathrel{-\!\!*} \mathsf{graph}(\gamma'')\big)} \tag{3}$$

Here we abuse notation a little bit. The conclusion of the "rule" (actually, lemma) is exactly right and appropriately generic, so spatial ramification lemmas of the kind given in §5 can handle the dirty spatial work for us. However, the second premise uses a notation for "mathematical graph node update" that is customized for union-find graphs, since most graphs have more than a rank and single outgoing edge. More seriously, updating a mathematical graph cannot be done willy-nilly; it is only defined when the properties that restrict the mathematical structure of γ are preserved. For example, in the case of union-find graphs, the graph must be acyclic (other than at roots). In Coq, these properties are carried around via dependent types, as will be explained in §4.1.

The example-specific challenge in proving `find` is showing that this update can be done properly, *i.e.* from $\mathsf{x} \in V(\gamma) \land \gamma(\mathsf{x}) = (r, pa)$ and $\mathsf{x} \neq pa \land \mathit{findS}(\gamma, pa, \gamma') \land \mathit{uf\_root}(\gamma', pa, rt)$ we must prove

$$\exists \gamma''.\ \gamma'' = [x \rightarrow (r, rt)]\gamma' \land \mathit{uf\_root}(\gamma'', \mathsf{x}, rt) \land \mathit{findS}(\gamma, \mathsf{x}, \gamma'')$$

This says: after compressing your parent and finding its root, compress yourself by rerouting your own parent pointer to your (soon-to-be former) parent's root. The existential in the goal is nontrivial exactly because the update $[x \rightarrow (r, rt)]\gamma'$ is not always kosher. This lemma requires some effort to prove, but is completely isolated from the grungy details of C. With the second localization block complete, the remainder of the verification is straightforward.

2.3 Our Seven Verified Examples

In addition to `find` shown above, we prove `union` in the same style, thus completing the verification of union-find for `malloc`-allocated nodes. We also verify a *second* version of union-find that uses arrays rather than nodes. The two programs look different spatially, but use exactly the same abstract mathematical definitions, *e.g.* for *findS*. This suggests that we have separated the abstract algorithmic reasoning from the specific details of heap representation, as we will explain in §7.

We discuss our verification of a graph `mark`ing algorithm in §3. We also verify a version of `mark` for directed acyclic graphs (DAGs), which is both easier and harder than marking cyclic graphs: we get genuine separation between the root and its children, but we also need to maintain acyclicity if we modify the link structure. We verify `copy`, a deep structure-preserving graph copying algorithm, which is tricky because we must initially reason about a "copy" that is under construction, but eventually show isomorphism with the original. As an example of more aggressive modifications to the link structure of a graph, we verify `spanning`, which prunes a graph into its spanning tree.

We elide discussions about `spanning`, DAG `mark`, and `copy` in the interest of space, but the verification code is in our artifact. Our flagship example, the CertiCoq garbage collector, is in §6.

3 LINKING EXISTENTIALS IN LOCALIZATIONS

In this section we first (§3.1) ensure our feet are solidly planted by proving why Localize is sound, and indeed equivalent to Frame. Second (§3.2), we address the bug from §2.2 and show that Localize is indeed strong enough to robustly handle modified local program variables. Third (§3.3), we showcase two additional features of our framework, linked existentials and a fold/unfold style for spatial graphs, by covering the verification of a program that marks a cyclic graph.

3.1 Soundness of Localize

In Figure 2 we put proof sketches that show that Localize and Frame are equivalent. They require a little care with quantifiers, but are in essence straightforward. In the latter proof set $R \triangleq F$, choose x_f fresh, and range the quantifiers over the unit type. Notice that in both directions the restriction on modified program variables is satisfied: in the first proof, Localize's side condition that $\mathsf{FreeVar}(R) \cap \mathsf{ModVar}(c) = \varnothing$ is exactly what Frame needs; in the second, Frame's side condition that $\mathsf{FreeVar}(F) \cap \mathsf{ModVar}(c) = \varnothing$ is exactly what Localize needs (since $R \triangleq F$). The equivalence between Frame and Localize means that our techniques will be sound in any separation logic.

Note on Notation. Localization blocks can safely nest. When the ramification entailment is not noteworthy we can omit the $\lightning(i)$ reference in pen-and-paper proofs. When we wish to save vertical space we write $\{G_1\} \searrow \{L_1\}$ and $\{G_2\} \swarrow \{L_2\}$. Since Localize can derive Frame, our notation also clarifies pen-and-paper uses of Frame, especially in multi-line contexts with nontrivial frame F.

$$\dfrac{G_1 \vdash L_1 * R \qquad \dfrac{\{L_1\}\ c\ \{\exists x.\ L_2\}}{\{L_1 * R\}\ c\ \{(\exists x.\ L_2) * R\}}\ \textsc{Frame} \qquad \dfrac{\dfrac{\vdots}{(\exists x.\ L_2) * (\forall x.\ (L_2 \mathrel{-\!\!*} G_2)) \vdash \exists x.\ G_2}\ \textsc{Tauto}}{(\exists x.\ L_2) * R \vdash \exists x.\ G_2}\ \textsc{Cut}\ (\dagger)}{\{G_1\}\ c\ \{\exists x.\ G_2\}}\ \textsc{Cons}$$

($\dagger$) $R \vdash \forall x.\ (L_2 \mathrel{-\!\!*} G_2)$ is a premise of Localize

$$\dfrac{\dfrac{\dfrac{\{P\}\ c\ \{Q\} \qquad Q \vdash \exists x_f.\ Q}{\{P\}\ c\ \{\exists x_f.\ Q\}}\ \textsc{Cons} \qquad F \vdash \forall x_f.\ (Q \mathrel{-\!\!*} (Q * F))}{\{P * F\}\ c\ \{\exists x_f.\ (Q * F)\}}\ \textsc{Localize} \qquad \exists x_f.\ (Q * F) \vdash Q * F}{\{P * F\}\ c\ \{Q * F\}}\ \textsc{Cons}$$

Fig. 2. Proving Localize from Frame, and conversely Frame from Localize

3.2 Smoothly Handling Modified Program Variables

Consider using Localize to verify the program below.

```
//{x = 5 ∧ A} ↘ {x = 5 ∧ B}
    ...; x = x + 1; ...;
//{x = 6 ∧ D} ↙ {x = 6 ∧ C}
```

Suppose that other (elided) lines of the program make localization desirable, even though it is overkill for a single assignment. The key issue is that if one sets $R \triangleq L_2 \mathrel{-\!\!*} G_2$ as we tried to do in §2.2, the program variable x appears in all four positions in the ramification entailment

$$\overbrace{(\mathsf{x}=5 \land A)}^{G_1} \vdash \overbrace{(\mathsf{x}=5 \land B)}^{L_1} * \Big(\overbrace{(\mathsf{x}=6 \land C)}^{L_2} \mathrel{-\!\!*} \overbrace{(\mathsf{x}=6 \land D)}^{G_2}\Big)$$

For the sake of simplicity, assume that in the above snippet only x is modified and that x does not appear free in A, B, C or D. Let us further assume that, modulo the local variable issue we are trying to solve, the entailment holds. In other words, let us assume that $A \vdash B * (C \mathrel{-\!\!*} D)$.

Turning to the local variable issue itself, in §2.2 we observed that $L_2 \mathrel{-\!\!*} G_2$ does **not** ignore the modified program variable x, preventing us from meeting Localize's side condition[3]. Intuitively, the side condition on Localize is a bit too strong since it prevents us from mentioning variables in the postconditions that have been modified by code c. As in other cases when life gets tough, what we need is an elegant little dance, and as with most dances, one should lead by example.

First, define $\hat{L_2}(x_f) \triangleq (x_f = 6 \land C)$ and $\hat{G_2}(x_f) \triangleq (x_f = 6 \land D)$, *i.e.* replace the troublesome program variable x in L_2 and G_2 with a harmless fresh metavariable x_f. Next, notice that with a carefully chosen existential quantifier, we can express the original L_2 with the new $\hat{L_2}$ while keeping the troublesome program variable x isolated. The new decorated program is below.

```
//{x=5 ∧ A} ↘ {x=5 ∧ B}
    ...; x = x + 1; ...;
//{x=6 ∧ C}
// ↙ {∃x_f. (x_f = x) ∧ (x_f=6 ∧ C)}
//{∃x_f. (x_f = x) ∧ (x_f=6 ∧ D)}
//{x=6 ∧ D}
```

[3] There is another problem: in the standard model for local variable treatment in separation logic, the separating implication is vacuously true since x cannot simultaneously be both 5 and 6. But since two fatal problems are overkill, let us move on.

Notice that lines 4 and 5 are exactly in the form $\exists x_f.\hat{\hat{L}}_2(x_f)$ and $\exists x_f.\hat{\hat{G}}_2(x_f)$, *i.e.* exactly in the format permitted by Localize, where $\hat{\hat{L}}_2(x_f) \triangleq (x_f = \mathsf{x}) \land \hat{L}_2(x_f)$, *i.e.* $(x_f = \mathsf{x}) \land (x_f = 6 \land C)$. $\hat{\hat{G}}_2(x_f)$ is similar. Now apply Localize with $R \triangleq \forall x_f.\ \hat{L}_2(x_f) \mathrel{-\!\!*} \hat{G}_2(x_f)$, *i.e.* $\forall x_f.\ (x_f = 6 \land C) \mathrel{-\!\!*} (x_f = 6 \land D)$. By construction, R is free from all program variables modified by c, so Localize's side condition is satisfied. All that remains is to prove Localize's two entailments. Let us consider them in reverse order. The second one is $R \vdash \forall x_f.\ \big(\hat{\hat{L}}_2(x_f) \mathrel{-\!\!*} \hat{\hat{G}}_2(x_f)\big)$, *i.e.*

$$\forall x_f.\ \big(\hat{L}_2(x_f) \mathrel{-\!\!*} \hat{G}_2(x_f)\big) \vdash \forall x_f.\ \Big(\big((\mathsf{x} = x_f) \land \hat{L}_2(x_f)\big) \mathrel{-\!\!*} \big((\mathsf{x} = x_f) \land \hat{G}_2(x_f)\big)\Big)$$

This is a long-winded tautology, and is automatic in a tool. The first entailment is $G_1 \vdash L_1 \mathrel{-\!\!*} R$, *i.e.*

$$(\mathsf{x} = 5 \land A) \vdash (\mathsf{x} = 5 \land B) * \big(\forall x_f.\ (x_f = 6 \land C) \mathrel{-\!\!*} (x_f = 6 \land D)\big)$$

This can be broken into the "variable-related" part $\mathsf{x} = 5 \vdash (\mathsf{x} = 5) * \big(\forall x_f.\ (x_f = 6 \mathrel{-\!\!*} x_f = 6)\big)$, which is also a tautology, and the "spatial" part $A \vdash B * (C \mathrel{-\!\!*} D)$, which is true by assumption above.

With careful engineering, this modified-variable dance can be done fully automatically, in a way that is hidden from end-users. The only remaining proof goal is the spatial part, which captures the key action of the localization block. In practice, we solve these via generic lemmas from §5.

Discussion. The delicacy and detail in the dance above may seem to be making mountains out of molehills, since a careful treatment of modified program variables is hardly a sexy topic. Indeed, in pen-and-paper systems they are molehills, with any number of workarounds including: making local program transformations to introduce fresh variables and arguing for program equivalence, using variables-as-resource [Bornat et al. 2006], or even just sweeping the issue under the rug.

These solutions are not viable when using existing toolsets in a mechanized context. Either we must reinvent a **very** large wheel—combined, VST and CompCert are about 840k LOC—or we must dance within their constraints. VST does not use variables as resource, nor does it have modules to reason about program equivalence. Moreover, most other mechanized verification systems [Beckert et al. 2007; Bengtson et al. 2012; Chin et al. 2010; Distefano and Parkinson 2008] do not support these solutions either. Because we respect these design decisions, our solutions can be incorporated more easily: in §7 we will see that our additions to VST are less than 1% of its codebase.

3.3 Linked Existentials

We have already seen that allowing existentials in postconditions lets us handle modified program variables properly. However, these "linked existentials"—recall that our previous technique hinged on the fact that the existential witness to the variable x_f in the local postcondition L_2 was carried over to the corresponding existential witness in the global postcondition—have other uses as well. To illustrate them, and to demonstrate other aspects of our system, in particular our ability to explore a graph recursively via fold/unfold, we consider another example.

In Figure 3 we put the code and proof sketch of the classic `mark` algorithm that visits and colors every reachable node in a heap-represented graph. The `mark` example contrasts from `find` in several respects. First, it modifies the labels of nodes instead of the edges. Second, each node has two outgoing edges rather than one, so the graph can have a more complex shape. Third, the graph can be nontrivially cyclic. Fourth, the specification we certify (lines 3 and 29) is *local* rather than *global*:

$$\{\mathsf{m\_graph}(\mathsf{x}, \gamma)\}\ \texttt{mark(x)}\ \{\exists \gamma'.\ \mathsf{m\_graph}(\mathsf{x}, \gamma') \land \mathit{mark}(\gamma, \mathsf{x}, \gamma')\}$$

The specification is again stated with mathematical γ, although $\gamma(\mathsf{x})$ now maps to triples (m, l, r) where m is a "mark" bit (0 or 1) and $\{l, r\} \subseteq V(\gamma) \uplus \{\texttt{null}\}$ are the neighbors of x. By "local", we mean that the predicate $\mathsf{m\_graph}(\mathsf{x}, \gamma)$ says that the heap represents *only the nodes in* γ *that are*

```
1   struct Node { int _Alignas(16) m;
2           struct Node * _Alignas(8) l, * r };
3   void mark(struct Node * x) {
4   // $\{\mathsf{m\_graph}(\mathtt{x}, \gamma)\}$
5     struct Node * l, * r; int root_mark;
6     if (x == 0) return;
7   // $\{\mathsf{m\_graph}(\mathtt{x}, \gamma) \wedge \exists m, l, r. \gamma(\mathtt{x}) = (m, l, r)\}$
8   // $\{\mathsf{m\_graph}(\mathtt{x}, \gamma) \wedge \gamma(\mathtt{x}) = (m, l, r)\}$
9   // $\searrow$ $\{\mathtt{x} \mapsto m, -, l, r\}$
10    root_mark = x -> m;
11  // $\swarrow$ $\{\mathtt{x} \mapsto m, -, l, r \wedge m = \mathtt{root\_mark}\}$
12  // $\{\mathsf{m\_graph}(\mathtt{x}, \gamma) \wedge \gamma(\mathtt{x}) = (m, l, r) \wedge m = \mathtt{root\_mark}\}$
13    if (root_mark == 1) return;
14  // $\{\mathsf{m\_graph}(\mathtt{x}, \gamma) \wedge \gamma(\mathtt{x}) = (0, l, r)\}$
15  // $\searrow$ $\{\mathtt{x} \mapsto 0, -, l, r \wedge \gamma(\mathtt{x}) = (0, l, r)\}$
16  $\lightning$(8)  l = x -> l; r = x -> r; x -> m = 1;
17  // $\swarrow$ $\left\{\begin{array}{l}\mathtt{x} \mapsto 1, -, \mathtt{l}, \mathtt{r} \wedge \gamma(\mathtt{x}) = (0, \mathtt{l}, \mathtt{r}) \wedge \\ \exists \gamma'. \mathit{mark1}(\gamma, \mathtt{x}, \gamma')\end{array}\right\}$
18  // $\{\exists \gamma'.\mathsf{m\_graph}(\mathtt{x}, \gamma') \wedge \gamma(\mathtt{x}) = (0, \mathtt{l}, \mathtt{r}) \wedge \mathit{mark1}(\gamma, \mathtt{x}, \gamma')\}$
19  // $\{\mathsf{m\_graph}(\mathtt{x}, \gamma') \wedge \gamma(\mathtt{x}) = (0, \mathtt{l}, \mathtt{r}) \wedge \mathit{mark1}(\gamma, \mathtt{x}, \gamma')\}$
20  // $\searrow$ $\{\mathsf{m\_graph}(\mathtt{l}, \gamma')\}$
21  $\lightning$(7)  mark(l);
22  // $\swarrow$ $\{\exists \gamma''.\mathsf{m\_graph}(\mathtt{l}, \gamma'') \wedge \mathit{mark}(\gamma', \mathtt{l}, \gamma'')\}$
23  // $\left\{\begin{array}{l}\exists \gamma''.\mathsf{m\_graph}(\mathtt{x}, \gamma'') \wedge \gamma(\mathtt{x}) = (0, \mathtt{l}, \mathtt{r}) \wedge \\ \mathit{mark1}(\gamma, \mathtt{x}, \gamma') \wedge \mathit{mark}(\gamma', \mathtt{l}, \gamma'')\end{array}\right\}$
24  // $\left\{\begin{array}{l}\mathsf{m\_graph}(\mathtt{x}, \gamma'') \wedge \gamma(\mathtt{x}) = (0, \mathtt{l}, \mathtt{r}) \wedge \\ \mathit{mark1}(\gamma, \mathtt{x}, \gamma') \wedge \mathit{mark}(\gamma', \mathtt{l}, \gamma'')\end{array}\right\}$
25  // $\searrow$ $\{\mathsf{m\_graph}(\mathtt{r}, \gamma'')\}$
26  $\lightning$(7)  mark(r);
27  // $\swarrow$ $\{\exists \gamma'''.\mathsf{m\_graph}(\mathtt{r}, \gamma''') \wedge \mathit{mark}(\gamma'', \mathtt{r}, \gamma''')\}$
28  // $\left\{\begin{array}{l}\exists \gamma'''.\mathsf{m\_graph}(\mathtt{x}, \gamma''') \wedge \gamma(\mathtt{x}) = (0, \mathtt{l}, \mathtt{r}) \wedge \\ \mathit{mark1}(\gamma, \mathtt{x}, \gamma') \wedge \mathit{mark}(\gamma', \mathtt{l}, \gamma'') \wedge \mathit{mark}(\gamma'', \mathtt{r}, \gamma''')\end{array}\right\}$
29  } // $\{\exists \gamma'''.\mathsf{m\_graph}(\mathtt{x}, \gamma''') \wedge \mathit{mark}(\gamma, \mathtt{x}, \gamma''')\}$
```

$$\begin{aligned}\mathsf{m\_graph}(x, \gamma) \;\Leftrightarrow\; & (x = 0 \wedge \mathsf{emp}) \vee \\ & \exists m, l, r.\ \gamma(x) = (m, l, r) \wedge x \bmod 16 = 0 \wedge \\ & x \mapsto m, -, l, r \mathbin{\cup\!\!\!\!\!\ast} \mathsf{m\_graph}(l, \gamma) \mathbin{\cup\!\!\!\!\!\ast} \mathsf{m\_graph}(r, \gamma)\end{aligned} \tag{4}$$

$$\mathit{mark1}(\gamma, x, \gamma') \triangleq \forall v. \gamma'(v) = \begin{cases}(1, l, r) & \text{when } x = v \wedge \\ & \gamma(v) = (0, l, r) \\ \gamma(v) & \text{otherwise}\end{cases}$$

$$v_1 \overset{\gamma}{\rightsquigarrow}_0 v_2 \triangleq \exists l, r.\ \gamma(v_1) = (0, l, r) \wedge v_2 \in \{l, r\}$$

$$v_1 \overset{\gamma}{\rightsquigarrow}{}_0^{\star}\, v_2 \triangleq \text{reflexive, transitive closure of “} \overset{\gamma}{\rightsquigarrow}_0 \text{”}$$

$$\mathit{mark}(\gamma, x, \gamma') \triangleq \forall v. \gamma'(v) = \begin{cases}(1, l, r) & \text{when } x \overset{\gamma}{\rightsquigarrow}{}_0^{\star}\, v \wedge \\ & \gamma(v) = (-, l, r) \\ \gamma(v) & \text{otherwise}\end{cases}$$

Fig. 3. Clight code and proof sketch for bigraph mark

reachable from x. Rather than passing the entire graph around as `find` does, `mark` uses the fold/unfold relationship given in (4) to "unfold" the graph as if it were an inductive predicate.

This fold/unfold relationship deserves attention. Note the use of the "overlapping conjunction" $\cup\!\!\!\!\!\ast$ of separation logic; informally $P \mathbin{\cup\!\!\!\!\!\ast} Q$ means that P and Q may overlap in the heap (*e.g.*, nodes in the left subgraph can also be in the right subgraph or even be the root x). The unspecified sharing indicated by the $\cup\!\!\!\!\!\ast$ connective[4] is part of why graph-manipulating algorithms are so hard to verify (*e.g.*, it is hard to apply the Frame rule). Second, (4) shows how industrial-strength settings complicate verification. Lines 1–2 define the data type `Node` used by `mark`. The `_Alignas(`n`)` directives tell CompCert to align fields on n-byte boundaries. As explained in §5.3, this alignment is necessary in C-like memory models to prove fold-unfold, which is why (4) includes an alignment restriction $x \bmod 16 = 0$ and an existentially-quantified "blank" second field for the root $x \mapsto m, -, l, r$. In our proofs, the alignment restriction and blank second field are neatly hidden behind the scenes.

Just as with `find`, the postcondition of `mark` is specified *relationally*, *i.e.* $\{\exists \gamma'.\ \mathsf{m\_graph}(\mathtt{x}, \gamma') \wedge \mathit{mark}(\gamma, \mathtt{x}, \gamma')\}$ instead of *functionally*, *i.e.* $\{\mathsf{m\_graph}\big(\mathtt{x}, \mathit{mark}(\gamma, \mathtt{x})\big)\}$. In the first case *mark* is a relation that specifies that γ' is the result of correctly marking γ from x, whereas in the second *mark* is a function that **computes** a new graph, which is the result of marking γ from x. A relational approach is better for both theoretical and practical reasons. Theoretically, relations are preferable because they are more general. For example, relations allow "inputs" to have no "outputs" (*i.e.* be partial) or alternatively have many outputs (*i.e.* be nondeterministic). Nondeterminism can be quite

[4] The standard semantics of the separation logic connectives used in this paper are in Figure 8 on page 13.

useful when specifying programs; for example, the CertiCoq garbage collector (§6) is specified nondeterministically to avoid, among other things, specifying how `malloc` allocates fresh blocks of memory. Relations are also preferable to functions because they are more compositional.

Practically, it is painful to define computational functions over graphs in a proof assistant like Coq. For example, Coq requires that all functions terminate—a nontrivial proof obligation over cyclic structures like graphs—but our verification of `mark` is only for partial correctness. Defining relations is much easier because *e.g.* one can use quantifiers and does not have to prove termination. The *mark* and *mark1* relations we use are defined straightforwardly at the bottom of Figure 3.

The highlights of the proof are as follows. In lines 9–11, imagine unfolding the m_graph predicate in line 8 using equation 4 and then zooming in to the root node x for lines 9–11, before zooming back out in line 12. Lines 19–23 of Figure 3 contain an example where we use the power of linked existentials in the Localize rule to "extract" the existentially-quantified γ'' from inside the localization block to outside it. The rest of the proof is relatively routine.

4 A REUSABLE LIBRARY OF FORMALIZED GRAPH THEORY

When verifying the functional correctness of graph algorithms it is natural to want to use graph theory to describe program behavior. Over the past 25 years, researchers in mechanized proofs have explored many choices for the critical definitions of graph theory (§8). Unfortunately, most such choices are not powerful enough, or expressive enough, to mechanically verify realistic algorithms written in C. Accordingly, an essential component of our work is a library of formalized graph theory that can support such verifications. As will be shown in §7, our mathematical graph constructions comprise a considerable fraction of our codebase, so it is vital that our framework be highly modular to enable reuse of definitions and proofs from one example to the next. In this section we present our mathematical graph framework with an emphasis on this modularity.

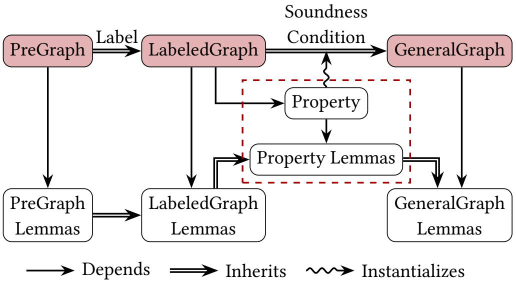

Fig. 4. Structure of the Mathematical Graph Library

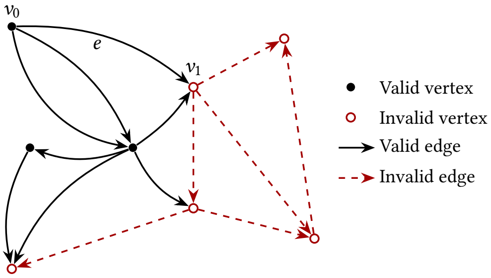

Fig. 5. PreGraph with valid/invalid vertices/edges.

4.1 Definitions of Graphs

Our first challenge is that graph theory is usually based on *set theory* but our formalization in Coq is based on *type theory*. Instead of formalizing set theory in Coq and then building graph theory atop of it, we formalize graph theory directly in Coq, as this lets us take advantage of Coq's built-in support for type-related constructions. To reconcile the dichotomy between a very general library and highly specialized examples, we develop our graphs gradually over three linked concepts: PreGraph, LabeledGraph and GeneralGraph. Figure 4 shows the architecture of the library.

PreGraph. A PreGraph is a hextuple ($\mathcal{V}$, $\mathcal{E}$, V, E, src, dst). Arguments $\mathcal{V}$ and $\mathcal{E}$ are the underlying carrier types of vertices and edges. V and E are predicates over $\mathcal{V}$ and $\mathcal{E}$ that specify the notion of *validity* in the graph. Finally, src and dst : $\mathcal{E} \to \mathcal{V}$ map each edge to its source and destination.

$$
\begin{aligned}
\text{path} &\triangleq (v_0, [e_0, e_1, \ldots, e_k]) \\
\mathtt{end}\big(\gamma, (v, [])\big) &\triangleq v \\
\mathtt{end}\big(\gamma, (v, [e_1, \ldots, e_n])\big) &\triangleq \mathtt{dst}_\gamma(e_n) \\
\mathtt{s\_evalid}(\gamma, e) &\triangleq \mathsf{E}_\gamma(e) \land \\
&\quad\; \mathsf{V}_\gamma(\mathtt{src}_\gamma(e)) \land \\
&\quad\; \mathsf{V}_\gamma(\mathtt{dst}_\gamma(e)) \\
\mathtt{valid\_path}\big(\gamma, (v, [])\big) &\triangleq \mathsf{V}_\gamma(v) \\
\mathtt{valid\_path}\big(\gamma, (v, [e_1, e_2, \ldots, e_n])\big) &\triangleq v = \mathtt{src}_\gamma(e_1) \land \\
&\quad\; \mathtt{s\_evalid}(\gamma, e_1) \land \\
&\quad\; \mathtt{dst}_\gamma(e_1) = \mathtt{src}_\gamma(e_2) \land \\
&\quad\; \mathtt{s\_evalid}(\gamma, e_2) \land \ldots \land \\
&\quad\; \mathtt{dst}_\gamma(e_{n-1}) = \mathtt{src}_\gamma(e_n)
\end{aligned}
$$

$$
\begin{aligned}
\gamma_1 \cong \gamma_2 &\triangleq \forall e.\, \mathsf{E}_{\gamma_1}(e) \Leftrightarrow \mathsf{E}_{\gamma_2}(e) \land \\
&\quad\; \forall v.\, \mathsf{V}_{\gamma_1}(v) \Leftrightarrow \mathsf{V}_{\gamma_1}(v) \land \\
&\quad\; \forall e.\, \mathsf{E}_{\gamma_1}(e) \Rightarrow \mathtt{src}_{\gamma_1}(e) = \mathtt{src}_{\gamma_2}(e) \land \\
&\quad\; \qquad\qquad\quad \mathtt{dst}_{\gamma_1}(e) = \mathtt{dst}_{\gamma_2}(e) \\
\gamma \models s \overset{p}{\rightsquigarrow} t &\triangleq \mathtt{valid\_path}(\gamma, p) \land \\
&\quad\; \mathtt{fst}(p) = s \land \mathtt{end}(\gamma, p) = t \\
\gamma \setminus S &\triangleq \Big(\mathcal{V}_\gamma, \mathcal{E}_\gamma, \mathtt{src}_\gamma, \mathtt{dst}_\gamma, \\
&\qquad\quad \lambda x.\, \gamma_{\mathsf{V}}(x) \land \neg S_{\mathsf{V}}(x), \\
&\qquad\quad \lambda x.\, \gamma_{\mathsf{E}}(x) \land \neg S_{\mathsf{E}}(x)\Big) \\
\mathit{reachable}(\gamma, x) &\triangleq \left\{t \mid \exists p\,.\, \gamma \models x \overset{p}{\rightsquigarrow} t\right\}
\end{aligned}
$$

Fig. 6. Some PreGraph definitions

The benefits of introducing validity are twofold. The first is a neat resolution of the incompatibility between type theory and set theory. In set theory, one element can belong to multiple sets, and adding or removing vertices or edges is as easy as altering the set directly to represent the result of the operation. In type theory, however, a term can only belong to one type, which makes it difficult to analogously change the type to represent the result. As is common practice, the predicates V and E specify whether a vertex/edge is *valid* (in the graph) or *invalid* (out). Adding or removing vertices/edges is as simple as weakening or strengthening these two predicates.

The second benefit is the ability to represent incomplete graphs. Consider starting from a graph γ and then removing a subgraph; the remaining "doughnut" structure is **not** necessarily a graph, since there may be dangling edges pointing into the "hole". Figure 5 shows just such a situation, where a connected graph (everything that is reachable from v_0) has had the connected subgraph reachable from v_1 removed, thus leaving the edge e dangling. The last conjunct in the *findS* relation from Figure 1 is an example of where a real verification needs to reason about just such a doughnut, in particular to specify that the unreachable portion of a graph has not changed.

We define many fundamental graph concepts on PreGraphs, including structures like *path**, predicates such as *is_cyclic* and *reachable**, operations such as *add_vertex* and *remove_edge*, and relations between PreGraphs such as *structurally_identical** and *subgraph*. Definitions of the concepts marked with asterisks are shown in Figure 6 to give a flavor of the subtleties involved in getting definitions that really work. These general concepts, together with around 500 derived lemmas, provide a solid foundation for more specific theorems needed in concrete verifications.

LabeledGraph. A LabeledGraph is septuple (PreGraph, $\mathcal{L}_\mathcal{V}$, $\mathcal{L}_\mathcal{E}$, $\mathcal{L}_\mathcal{G}$, `vl`, `el`, `gl`) that augments a PreGraph with *labels* on vertices, edges, and/or the graph as a whole. $\mathcal{L}_\mathcal{V}$, $\mathcal{L}_\mathcal{E}$, and $\mathcal{L}_\mathcal{G}$ are the associated carrier types, and `vl`, `el`, and `gl` are the labeling functions themselves. Many classic graph problems, from union-find (node ranks) to Dijkstra (edge weights), require such labels. The need for a label on the graph as a whole is not as obvious; in §6 we use one in the garbage collector to keep track of the number of generations and their boundaries. Since every LabeledGraph is built on a PreGraph, it inherits all of the PreGraph's lemmas via Coq's *type coercion* mechanism while also opening the doors to additional lemmas involving its own labels.

GeneralGraph. PreGraphs and LabeledGraphs let us state and prove many useful lemmas that follow essentially by the nature of our graph constructions. However, when proving the correctness of graph algorithms, we often need more specificity in our mathematical graphs so that we may model the real program's behaviors closely. For example, the `uf_graph` used in `find` restricts each vertex to having exactly one out-edge. On the other hand, these restrictions vary greatly by algorithm, so we do not want to bake them into our core definitions. We achieve this flexibility

$$\mathrm{MathGraph}(\gamma) \triangleq \left\{\begin{array}{r}\texttt{null} : V\\ \texttt{valid\_graph} : \forall e \,.\, \texttt{evalid}(\gamma, e) \Rightarrow \texttt{vvalid}\big(\gamma, \texttt{src}(\gamma, e)\big) \land \big(e = \texttt{null} \lor \texttt{vvalid}(\gamma, e)\big)\\ \texttt{valid\_not\_null} : \forall v \,.\, \texttt{vvalid}(\gamma, v) \Rightarrow v \neq \texttt{null}\end{array}\right\}$$

$$\mathrm{LstGraph}(\gamma) \triangleq \left\{\begin{array}{r}\texttt{out} : V \rightarrow E\\ \texttt{only\_one\_edge} : \forall v,\ e \,.\, \texttt{vvalid}(\gamma, v) \Rightarrow \big(\texttt{src}(\gamma, e) = v \land \texttt{evalid}(\gamma, e)\big) \Leftrightarrow e = \texttt{out}(v)\\ \texttt{acyclic\_path} : \forall v,\ p \,.\, \gamma \models v \overset{p}{\leadsto} v \Rightarrow p = (v, [])\end{array}\right\}$$

$$\mathrm{FiniteGraph}(\gamma) \triangleq \left\{\begin{array}{l}\texttt{finite\_v} : \exists S_v,\ n.\ \ |S_v| \leq n \land \forall v \,.\, \texttt{vvalid}(\gamma, v) \Rightarrow v \in S_v\\ \texttt{finite\_e} : \exists S_e,\ n.\ \ |S_e| \leq n \land \forall e \,.\, \texttt{evalid}(\gamma, e) \Rightarrow e \in S_e\end{array}\right\}$$

Fig. 7. Some GeneralGraph definitions

using GeneralGraphs, which augment LabeledGraphs by adding arbitrarily complex "soundness conditions", indicated in Figure 4 with a dashed border. Further, the type coercion we described earlier continues to apply, meaning that a GeneralGraph can seamlessly behave like its internal LabeledGraph or a PreGraph, thereby inheriting their lemmas. This combination of specificity and generality makes GeneralGraphs versatile. Moreover, we can compose complicated soundness conditions from reusable pieces, further enabling code sharing between algorithms.

4.2 Composing Soundness Plugins

Soundness conditions are often specific to each algorithm, but they feature some recurring themes. We take advantage of this pattern by developing *soundness plugins*, *i.e.* definitions of soundness conditions along with related lemmas. By combining these plugins we can describe the soundness condition we need for a particular algorithm. When proving lemmas about the resulting combination, we can use known facts about the separate plugins, in addition to lemmas that emerge due to the various combinations. This complexity is managed smoothly by Coq's typeclass system, increasing the compositionality of the system. Consider the following oft-used graph properties:

- BiGraph: there are exactly two outgoing edges per vertex
- LstGraph*: the graph is structured like a list, meaning that every vertex has one outgoing edge, and there no loops except trivial self-loops of length 0, signifying the end of the "list"
- MathGraph*: every valid edge has a valid source vertex, and its destination vertex is either valid or is a special invalid vertex called null
- FiniteGraph*: the sets of valid vertices and edges are both finite

Definitions of the concepts marked with asterisks are shown in Figure 7 for illustration. We can compose LstGraph, MathGraph, and FiniteGraph together into a new plugin called LiMaFin, which, incidentally, is the soundness condition of mathematical uf_graph we used to verify `find` in Figure 1. In our verification of `mark` in Figure 3, we use a similar soundness condition BiMaFin, which uses BiGraph instead of LstGraph. The commonalities and differences between LiMaFin and BiMaFin are readily apparent from their construction, and can be exploited for proof reuse.

5 DEFINING AND REASONING ABOUT SPATIAL GRAPHS

To prove the functional correctness of graph-manipulating algorithms implemented in a real language, we need to (1) connect the heap representation of graphs to the memory model of the programming language, and (2) connect the memory model to the mathematical properties of abstract graphs from §4. The first of these challenges turns out to be surprisingly subtle: the standard tactic of leveraging the Frame rule works well for tree-manipulating programs, but fails for graph-manipulating programs. We review the standard treatment for trees (§5.1), point out the issue with graphs (§5.2), and propose a fix for this (§5.3). The main challenge thereafter is to engineer a framework that is generic and modular enough to be useful in a variety of settings (§5.4).

$$\begin{array}{lcl}\sigma \models P * Q & \triangleq & \exists \sigma_1, \sigma_2.\, (\sigma_1 \oplus \sigma_2 = \sigma) \wedge (\sigma_1 \models P) \wedge (\sigma_2 \models Q) \\ \sigma \models P \mathrel{-\!\!*} Q & \triangleq & \forall \sigma_1, \sigma_2.\, (\sigma_1 \oplus \sigma = \sigma_2) \wedge (\sigma_1 \models P) \Rightarrow (\sigma_2 \models Q) \\ \sigma \models P \uplus\!\!\!\!* \; Q & \triangleq & \exists \sigma_1, \sigma_2, \sigma_3.\, (\sigma_1 \oplus \sigma_2 \oplus \sigma_3 = \sigma) \wedge (\sigma_1 \oplus \sigma_2 \models P) \wedge (\sigma_2 \oplus \sigma_3 \models Q)\end{array}$$

Fig. 8. Separation logic connectives; $\oplus$ is the join operation on states, *e.g.* a disjoint union on heaps

5.1 Separation Logic in Tree-Manipulating Programs

Figure 8 shows the standard semantic models for the separation logic connectives using a join relation $\oplus$ on an underlying separation algebra [Dockins et al. 2009]. In recent works [O'Hearn et al. 2001; O'Hearn 2012] that employ separation logic to give precise specifications of tree copy programs, the spatial representation of a binary tree is defined as a recursive predicate with an additional parameter: the mathematical tree τ.

$$\begin{aligned}\mathsf{tree}(x, \tau) \triangleq & \Big(x = 0 \wedge \mathsf{isatom}(\tau) \wedge \mathsf{emp}\Big) \vee \\ & \Big(\exists\, l, r, \tau_1, \tau_2.\ \tau = \langle \tau_1, \tau_2 \rangle \wedge (x \mapsto l, r) * \mathsf{tree}(l, \tau_1) * \mathsf{tree}(r, \tau_2)\Big)\end{aligned} \tag{5}$$

The mathematical tree τ encodes the shape of a binary tree. It is either an atom or a pair of tree encodings. For example, $\langle \mathsf{atom}, \langle\langle \mathsf{atom}, \mathsf{atom}\rangle, \mathsf{atom}\rangle\rangle$ is a valid encoding. A $\mathsf{tree}(x, \tau)$ is either an emp or $(x \mapsto l, r) * \mathsf{tree}(l, \tau_1) * \mathsf{tree}(r, \tau_2)$ where τ_1 and τ_2 are the left and right subtrees of τ. If τ_1 is atom then l must be null pointer and $\mathsf{tree}(l, \tau_1)$ must be emp. Otherwise $\mathsf{tree}(l, \tau_1)$ can be expanded recursively as above. The left part of Figure 9 shows a binary tree, and it should be straightforward to see that the data layout matches τ (the shape of the tree) exactly. In the verification of a tree copy program, the precondition is $\mathsf{tree}(x, \tau)$ and the postcondition is $\mathsf{tree}(x, \tau) * \mathsf{tree}(y, \tau)$. Having the same τ in the pre- and postconditions of the specification indicates that the program creates an exact clone of the original tree, not an arbitrarily-shaped tree.

This is an ideal use-case for the separation logic because the root, the left subtree, and the right subtree are disjoint by construction. By applying the Frame rule, one can safely reason about a particular $*$-separated branch of the tree and later compose the conclusion back into the rest of the tree without worrying about the effect on the rest of the tree. In the figure we show that manipulating the subtree under root v does not affect the rest of the tree.

Graphs are more complicated. Consider the right part of Figure 9, which shows a bigraph. Defining a recursive predicate in the style of (5), *i.e.* using the separating conjunction $*$, is a bad idea: a node can be shared arbitrarily, and so finding a suitable frame is difficult. In the figure we show that manipulating the subgraph reachable from w causes an unexpected node to be affected.

5.2 Recursive Definitions Yield Poor graph Predicates

Recursive predicates are ubiquitous in separation logic—so much so that when one writes the definition of a predicate as P "$\triangleq$" $\ldots P \ldots$, no one raises an eyebrow despite the dangers of circularity in mathematics. Indeed, the vast majority of the time there is no danger thanks to the magic of the

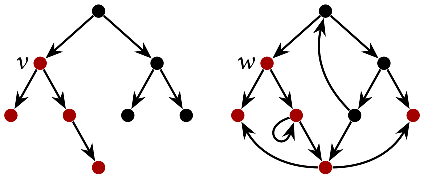

Fig. 9. A Binary Tree and a Binary Graph

address	value
102	0
101	100
100	42

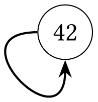

Fig. 10. A One-Cell Graph and its Heap Layout

$$\dfrac{\dfrac{100 \mapsto 42, 100, 0 \;\vdash\; 100 \mapsto 42, 100, 0 \uplus \mathsf{graph}_T(100, \hat{\gamma})}{100 \mapsto 42, 100, 0 \;\vdash\; \hat{\gamma}(100) = (42, 100, 0) \;\wedge\; 100 \mapsto 42, 100, 0 \uplus \mathsf{graph}_T(100, \hat{\gamma}) \uplus \mathsf{graph}_T(0, \hat{\gamma})}\,\ddagger}{100 \mapsto 42, 100, 0 \;\vdash\; \mathsf{graph}_T(100, \hat{\gamma})}\,\dagger$$

† Unfold graph_T, dismiss first disjunct (contradiction), introduce existentials (which must be 42,100,0)

‡ simplify using $P * \mathsf{emp} \dashv\vdash P$ and remove pure conjunct

Fig. 11. An attempt to prove a "simple" entailment

Knaster-Tarski fixpoint μ_{T} [Tarski 1955]. Formally, one does not define P directly, but rather defines a functional $F_P \overset{\triangle}{=} \lambda P.\ \ldots P \ldots$ and then defines P itself as $P \overset{\triangle}{=} \mu_{\mathrm{T}}\, F_P$. Assuming, as one typically does without comment, that F_P is *covariant*, i.e. $(P \vdash Q) \Rightarrow (F\,P \vdash F\,Q)$, one then enjoys the fixpoint equation $P \Leftrightarrow \ldots P \ldots$, formally justifying the typically-written pseudodefinition ("$\overset{\triangle}{=}$").

Suppose we define a graph predicate graph_T along the lines of the fold/unfold definition in Figure 3. Such a definition would use the overlapping conjunction $P \uplus Q$ (*c.f.* Figure 8) as follows:

$$\mathsf{graph}_T(x, \gamma) \overset{\triangle}{=} \Big(x = 0 \wedge \mathsf{emp}\Big) \vee \\ \Big(\exists m, l, r.\ \gamma(x) = (m, l, r) \wedge \big(x \mapsto m, l, r \uplus \mathsf{graph}_T(l, \gamma) \uplus \mathsf{graph}_T(r, \gamma)\big)\Big)$$

Although we can apply Knaster-Tarski (because the functional needed to define graph_T is covariant), the result is hard to use. Consider the memory m for a toy machine, where $m \models 100 \mapsto 42, 100, 0$. It seems clear, however, that this memory also represents a one-cell cyclic graph as illustrated in Figure 10, *i.e.* we want $m \models \mathsf{graph}_T(100, \hat{\gamma})$, where $\hat{\gamma}(100) = (42, 100, 0)$. This is equivalent to wanting to be able to prove $100 \mapsto 42, 100, 0 \vdash \mathsf{graph}_T(100, \hat{\gamma})$. Unfortunately, as illustrated in Figure 11, this is rather difficult to do since applying the natural proof techniques actually strengthens the goal. Part of the problem is that the recursive structure interacts very badly with $\uplus$: if the recursion involved $*$ then it **would** be provable, by induction on the finite memory (each "recursive call" would be on a strictly smaller subheap). This is why Knaster-Tarski works so well with list, tree, and DAG predicates in separation logic. We do not know if this entailment is provable, but the difficulties encountered in proving what "should be" straightforward suggest that Knaster-Tarski should be treated with caution when defining spatial predicates for graphs.

The other direction, $\mathsf{graph}_T(100, \hat{\gamma}) \vdash 100 \mapsto 42, 100, 0$, **is** true but is not easy to prove, relying on the constructions in §5.3 and the fact that μ_{T} constructs the least fixpoint. In contrast, $\mathsf{graph}_T(100, \hat{\gamma}) \vdash 100 \mapsto 42, 100, 0 * \top$ is easy.

Appel and McAllester proposed another fixpoint μ_{A} that is sometimes used to define recursive predicates in separation logic [Appel and McAllester 2001]. This time the functional F_P needs to be *contractive*, which to a first order of approximation means that all recursion needs to be guarded by $\triangleright$, the "approximation modality" [Appel et al. 2007], *i.e.* our graph predicate would look like

$$\mathsf{graph}_A(x, \gamma) \overset{\triangle}{=} (x = 0 \wedge \mathsf{emp}) \vee \exists m, l, r.\ \gamma(x) = (m, l, r) \wedge \\ x \mapsto m, l, r \uplus \triangleright\mathsf{graph}_A(l, \gamma) \uplus \triangleright\mathsf{graph}_A(r, \gamma)$$

This functional is contractive, and so the fixpoint is well-defined. Nevertheless, this definition has a subtle flaw: the resulting graph predicate is **not** "precise" in the separation logic sense:

$$precise(P) \overset{\triangle}{=} (\sigma_1 \models P) \Rightarrow (\sigma_2 \models P) \Rightarrow (\sigma_1 \oplus \sigma_1' = \sigma) \Rightarrow (\sigma_2 \oplus \sigma_2' = \sigma) \Rightarrow \sigma_1 = \sigma_2$$

The flaw stems from the fact that $\triangleright P$ is not precise for any P (at time zero, $\triangleright P$ is just $\top$), so graph_A is not precise either. The approximation modality's universal imprecision has never been noticed before in the literature. Accordingly, graph_A is also rather difficult to use.

5.3 Defining a Good graph Predicate

Rather than defining graph as a recursive fixpoint, we give it a flat structure. Graphs in separation logic have been defined in similar ways before, *e.g.* Sergey et al. [2015] and Charguéraud and Pottier [2019]; our innovation is that we prove—with the amount of precision required to convince Coq—that we still enjoy fold/unfold with our flat definition. We start with the iterated separating conjunction or "big star", which is first defined over lists and then extended to sets as follows:

$$\mathop{\Large\ast}_{\{l_1,\ldots,l_n\}} P \triangleq P(l_1) * P(l_2) * \ldots * P(l_n) \quad\Big|\quad \mathop{\Large\ast}_{S} P \triangleq \exists L.\ (\mathsf{NoDup}\ L) \wedge (\forall x.\ x \ \mathsf{in}\ L \Leftrightarrow x \in S) \wedge \mathop{\Large\ast}_{L} P$$

We are now ready to define a good graph predicate: $\mathsf{graph}(x,\gamma) \triangleq \mathop{\Large\ast}\limits_{v \in reachable(\gamma,x)} v \mapsto \gamma(v).$

The graph γ here need not be a bigraph, but *e.g.* can have many edges.

Our definition of graph is flat in the sense that there is no obvious way to follow the link structure recursively. Happily, we can recover a general recursive fold/unfold (if $x \mapsto \gamma(x)$ and the GeneralGraph has the necessary properties in its soundness condition):

$$\mathsf{graph}(x,\gamma) \Leftrightarrow x \mapsto \gamma(x) \uplus \Big(\biguplus_{n \in \mathsf{neighbors}(\gamma,x)} \mathsf{graph}(\gamma,n)\Big) \quad \text{where} \quad \biguplus_{l_1,\ldots,l_n} P \triangleq P(l_1) \uplus \ldots \uplus P(l_n) \tag{6}$$

The proof of the $\Leftarrow$ direction requires care. The difficulty is that if two nodes $x \mapsto \gamma(x)$ and $x' \mapsto \gamma(x')$ are *skewed*, *i.e.* "partially overlapping" with some—but not all—of x's memory cells shared with x', then the $\mathop{\Large\ast}$ on the left hand side cannot separate them. To avoid skewing we require that $x \mapsto \gamma(x)$ be *alignable*. A predicate P is alignable when

$$\forall x, y.\ \Big(P(x) \uplus P(y) \vdash \big(P(x) \wedge x = y\big) \vee \big(P(x) * P(y)\big)\Big)$$

That is, they overlap either completely or not at all. In a Java-like memory model this property is automatic because pointers in such a model always point to the root/beginning of an object. In contrast, in a C-like memory model such as in VST/CompCert, this property is not automatic because pointers can point anywhere. In such a model, alignment is most easily enforced by storing graph nodes at addresses that are multiples of an appropriate size (*e.g* this size is 16 in Figure 3).

Some of our VST proofs, *e.g.* for `find`, do not use fold/unfold, instead preferring to use the lemmas in §5.4 directly. Others, *e.g.* `mark`, do, and it is helpful to have both options avilable. We also prove fold/unfold lemma for DAGs in which we get a clean $*$ between the root and its $\uplus$-joined neighbors, rather than the $\uplus$ present in (6).

5.4 Ramification Libraries

Many tools (*e.g.* Charge! [Bengtson et al. 2012], Smallfoot [Berdine et al. 2005], jStar [Distefano and Parkinson 2008], HIP/SLEEK [Chin et al. 2010]) use the Direct Model to represent spatial predicates on the heap, but VST employs an unusually complex Step-Indexed heap model in order to support an unusually rich program logic [Appel et al. 2014]. Figure 12 shows the architecture of our spatial development. We will explain our two interfaces momentarily, but for now observe that both heap models can instantiate both interfaces. That is to say, we do not rely on any of the bells, whistles, or specialized properties that are only available in the Step-Indexed Model.

We modularize our spatial library over two interfaces: Core Logic and Supplementary Logic. Each interface defines some operators of separation logic along with relevant axioms to show how they work. For example, the definitions of $*$ and $-\!\!*$ are in Core Logic, along with key axioms and

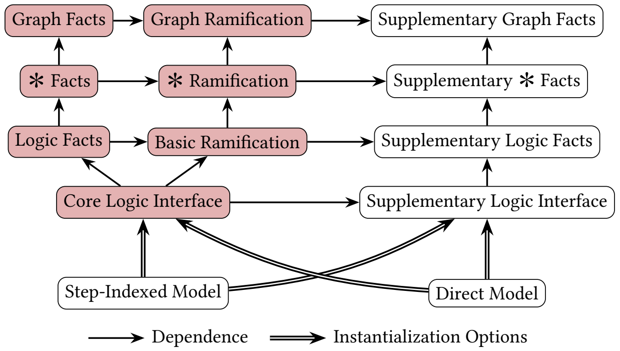

Fig. 12. Infrastructure of ramification library

rules such as $(P \vdash Q -\!\!* R) \Leftrightarrow (P * Q \vdash R)$. On the other hand, lesser-used operators such as $\mathbin{\cup\!\!\!\ast}$ and $-\!\circledast$ are in Supplementary Logic, along with rules such as $P \vdash P \mathbin{\cup\!\!\!\ast} P$. Generally speaking, our VST proofs only need Core properties (shaded in the figure) to prove our examples.

Above the Logic layer we have three towers, each three levels high. The tower on the left contains basic lemmas about Logic, $\mathop{\Large\ast}$, and graph. For instance, in the $\mathop{\Large\ast}$ Facts box we prove:

$$\frac{A \cap B = \varnothing}{\mathop{\Large\ast}_{x \in A} P(x) * \mathop{\Large\ast}_{x \in B} P(x) \Leftrightarrow \mathop{\Large\ast}_{x \in A \cup B} P(x)}$$

The middle tower is more interesting in that it is entirely focused on ramification entailments. A robust library of ramification entailments is essential to make ramification work smoothly in practice. The Basic Ramification box in the lower layer contains lemmas like the one below, which we use to break large ramification entailments into compositionally manageable pieces.

$$\frac{G_1 \vdash L_1 * \forall x.\ (L_2 -\!\!* G_2) \qquad G_1' \vdash L_1' * \forall x.\ (L_2' -\!\!* G_2')}{G_1 * G_1' \vdash (L_1 * L_1') * \forall x.\ \big((L_2 * L_2') -\!\!* (G_2 * G_2')\big)}$$

The $\mathop{\Large\ast}$ Ramification box in the middle layer contains lemmas like the one below:

$$\frac{A \cap B = \varnothing \qquad A' \cap B = \varnothing}{\mathop{\Large\ast}_{x \in A \cup B} P(x) \vdash \mathop{\Large\ast}_{x \in A} P(x) * \Big(\mathop{\Large\ast}_{x \in A'} P(x) -\!\!* \mathop{\Large\ast}_{x \in A' \cup B} P(x) \Big)}$$

The above lemma expresses large-scale replacement, clearing the way to cleanly establish a key graph-specific fact: Lemma 7, which is placed in the Graph Ramification box (top layer), is used on lines 21 and 26 of Figure 3 to reason about smoothly replacing subgraphs.

$$\frac{n \in \mathsf{neighbors}(\gamma, x)}{\mathsf{graph}(x, \gamma) \vdash \mathsf{graph}(n, \gamma) * \forall \gamma'.\mathit{mark}(\gamma, n, \gamma') \to \big(\mathsf{graph}(n, \gamma') -\!\!* \mathsf{graph}(x, \gamma')\big)} \tag{7}$$

Similarily, the same box contains the key lemma used on line 16 in Figure 3 to update one node:

$$\frac{\forall x_0 \neq x.\ \gamma(x_0) = \gamma'(x_0) \mathsf{neighbors}(\gamma, x) = \mathsf{neighbors}(\gamma', x)}{\mathsf{graph}(x, \gamma) \vdash x \mapsto \gamma(x) * \big(x \mapsto \gamma'(x) -\!\!* \mathsf{graph}(x, \gamma')\big)} \tag{8}$$

This layered structure enables proof reuse. All of the theorems for graph are proved from the properties of iterated separating conjunction, but having a modular library allows $\mathop{\Large\ast}$ to be reused in other structures smoothly. Further, all our verifications of different graph algorithms use the proof rules of graph at the top level in the library.

The Supplementary tower contains properties not used by most of the VST examples. This includes the fold/unfold relationship that we moved away from in §5.3, facts about precision, *etc.* These are currently included mostly for completeness, but do make our library more general should we wish to accommodate an alternate prover that needs separation logic facts about $\uplus\!\!\!\!\ast$, $-\!\!\circledast$, *etc.*

6 CERTIFYING A GARBAGE COLLECTOR FOR CERTICOQ

The CertiCoq compiler [Anand et al. 2017] translates Gallina code to Clight, which CompCert [Leroy 2006] then compiles to assembly. Gallina assumes infinite heap memory but Clight has a finite heap, so CertiCoq supports Gallina's assumption via memory management at the Clight level. In particular, the Clight code generated by CertiCoq contains calls to a garbage collector (GC), also written in Clight. CertiCoq aims to be end-to-end certified, so the GC must also be certified. We explain the code's operation (§6.1), abstract the problem to mathematical graphs (§6.2), explain two key functions from the code (§6.3, §6.4), and review interesting issues we found and resolved (§6.5).

6.1 Overview of the GC Program

The 12-generation GC, written in the spirit of the OCaml GC, is realistic and sophisticated, though by no means industrial-strength. Because CertiCoq uses OCaml's representation of blocks and values [Hickey et al. 2014], the GC must support features such as variable-length memory objects, object fields that may be boxed or unboxed and must be disambiguated at runtime, and pointers to places outside the GC-managed heap. That said, its task is easier than the OCaml GC's because its mutator is purely functional[5]. The mutator maintains an array of local variables called `args`, and the GC scans this array to calculate the root set. The GC collects the first generation into the second using Cheney's algorithm [Cheney 1970]. This may recursively trigger the collection of the second generation into the third, *etc.*, following which the GC returns control to the mutator. We provide further details about the GC's operation in our extended online paper.

The mutator's `args` array is critical to the GC's formal specification. The heap may change dramatically, but all heap memory objects that the mutator could reach by recursively following the fields of `args` *before* the collection must still be reachable, via similar steps, *after* the collection. This problem can be abstracted into mathematical graphs, where we must prove graph isomorphism.

6.2 From Clight to Mathematical Graphs

In the code, a generation is a contiguous memory segment. Three pointers capture key metainformation about a generation: `start` marks the first address, `limit`, the last address, and `next`, the next address available for allocation. Initially, `next = start`. New memory is added in a contiguous chunk starting at `next`, and `next` is incremented appropriately. The generation is full when `next = limit`. Metainformation about all 12 generations is stored in a 12-element array called `heap`.

The basic unit manipulated by the GC is a contiguous piece of memory called a *block*, which is a 22-bit header followed by an arbitrary-length array of *fields*. The length of a block's field array is stored in its header. Each field is either an unboxed integer data value or a pointer. To disambiguate the two, we follow OCaml's practice of requiring that all integers are odd (essentially, are only 31 bits long) and all pointers are even-aligned [Hickey et al. 2014]. Pointers may point either into the GC-controlled heap or at external structures outside the GC's purview.

Moving towards a mathematical abstraction, the 12 generations can be seen as a graph γ. Blocks are vertices, and pointers to other blocks are edges to other vertices. Vertices are tuples of the

[5] That is: Gallina is purely functional, and the Clight code generated by CertiCoq preserves this behavior.

form (g, i), meaning the vertex represents the the i^{th} block in the g^{th} generation. An edge is a tuple (v, i), meaning that the associated pointer is the i^{th} field of the block corresponding to vertex v. Each vertex is labeled with its integer data, along with metainformation such as whether it has already been forwarded, and if so, where its forwarded copy is. There is also a global label on the entire graph γ which has the start/limit addresses and number of vertices of each generation.

6.3 Forward

```
// $\left\{\begin{array}{l}\forall \gamma, \mathit{finf}, \mathit{tinf}, \mathit{from}, \mathit{to}, v, n.\\ \mathrm{gc\_graph}(\gamma) * \mathrm{finf}(\mathit{finf}) * \mathrm{tinf}(\mathit{tinf}) \wedge\\ \mathit{compat}(\gamma, \mathit{finf}, \mathit{tinf}, \mathit{from}, \mathit{to}) \wedge\\ \mathtt{s} = \mathit{start}(\gamma, \mathit{from}) \wedge \mathtt{l} = \mathtt{s} + \mathit{gensz}(\gamma, \mathit{from}) \wedge\\ \mathtt{n} = \mathit{nxtaddr}(\mathit{tinf}, \mathit{to}) \wedge \mathtt{p} = \mathit{vaddr}(\gamma, v) + n\end{array}\right\} \triangleq \phi_1$
void forward (value *s, *l, **n, *p) {
  value *v; value va = *p;
  if(Is_block(va)) {   //is ptr
  v = (value *)((void *)va);
  if(Is_from(s, l, v)) {   //in from
// $\left\{\begin{array}{l}\phi_1 \wedge \exists e, v'.\ \mathit{lab}(\gamma, v)[n] = e \wedge\\ \mathit{dst}(\gamma, e) = v' \wedge \mathtt{v} = \mathit{vaddr}(\gamma, v')\end{array}\right\} \triangleq \phi_7$
//↘ $\left\{\begin{array}{l}\exists \mathit{flds}', \mathit{hdr}'.\ \mathit{flds}' = \mathit{lab}(\gamma, v') \wedge\\ v' \mapsto \mathit{flds}' \wedge \mathit{hdr}' = \mathit{flds}'[-1]\end{array}\right\} \triangleq \phi_8$
    header_t hd = Hd_val(v);
//↙ $\{\phi_8 \wedge \mathtt{hd} = \mathit{val}(\mathit{hdr}')\}$
// $\{\phi_7 \wedge \mathtt{hd} = \mathit{val}(\mathit{hdr}')\}$
    if(hd == 0) {   //already forwarded
// $\{\phi_7 \wedge \mathtt{hd} = 0\} \triangleq \phi_{13}$
//↘ $\left\{\begin{array}{l}\exists \mathit{flds}, \mathit{flds}'.\ \mathtt{v} \mapsto \mathit{flds} \wedge v' \mapsto \mathit{flds}' \wedge\\ \mathit{flds} = \mathit{lab}(\gamma, v) \wedge \mathit{flds}' = \mathit{lab}(\gamma, v') \wedge\\ \mathit{flds}'[0] = \mathit{vaddr}(\gamma, \mathit{copy}(\gamma, v')) \wedge\\ \mathtt{p} = \&\mathit{flds}[n]\end{array}\right\} \triangleq \phi_{14}$
      *p = Field(v,0);
//↙ $\{\phi_{14} \wedge \mathit{flds}[n] := \mathit{flds}'[0]\}$
// $\left\{\begin{array}{l}\phi_{13} \wedge \exists \gamma'.\ \mathrm{gc\_graph}(\gamma') \wedge\\ \gamma' = \mathit{upd\_edge}(\gamma, e, \mathit{copy}(\gamma, v')) \wedge\\ \mathit{fwd\_postcondition}(\gamma, \gamma', \mathit{tinf}, \mathit{finf}, \mathit{from}, \mathit{to}, v, n)\end{array}\right\}$
    } else {   //not yet forwarded
// $\{\phi_7 \wedge \mathtt{hd} \neq 0\} \triangleq \phi_{19}$
      int i; int sz; value *new;
      sz = size(hd); new = *n+1;
      *n = new+sz;
// $\left\{\begin{array}{l}\phi_{19} \wedge \mathtt{sz} = \mathit{blocksize}(\mathtt{hd}) \wedge\\ \mathtt{new} = \mathit{start}(\gamma, \mathit{to}) + \mathit{used}(\gamma, \mathit{to}) + 1 \wedge\\ \mathtt{n} = \mathtt{new} + \mathtt{sz}\end{array}\right\} \triangleq \phi_{23}$
      Hd_val(new) = hd;
      for(i = 0; i < sz; i++)
      Field(new, i) = Field(v, i);
// $\left\{\begin{array}{l}\phi_{23} \wedge \exists \gamma', v'', \mathit{tinf}'.\\ \mathrm{gc\_graph}(\gamma') * \mathrm{tinf}(\mathit{tinf}') \wedge\\ v'' = \mathit{newly\_copied\_vertex}(\gamma, \mathit{to}) \wedge\\ \gamma' = \mathit{copy\_vertex}(\gamma, \mathit{to}, v', v'') \wedge\\ \mathit{compat}(\gamma', \mathit{finf}, \mathit{tinf}', \mathit{from}, \mathit{to})\end{array}\right\} \triangleq \phi_{27}$
      Hd_val(v) = 0;
      Field(v, 0) = (value)((void *)new);
// $\{\phi_{27} \wedge \mathit{val}(\mathit{hdr}') = 0 \wedge \mathit{flds}'[0] = \mathit{copy}(\gamma, v')\} \triangleq \phi_{30}$
//↘ $\{\exists \mathit{flds}.\ \mathtt{v} \mapsto \mathit{flds} \wedge \mathit{flds} = \mathit{lab}(\gamma', v)\} \triangleq \phi_{31}$
      *p = (value)((void *)new);
//↙ $\{\phi_{31} \wedge \mathit{flds}[0] := \mathit{vaddr}(\gamma, v'')\}$
// $\left\{\begin{array}{l}\phi_{30} \wedge \exists \gamma''.\ \mathrm{gc\_graph}(\gamma'') \wedge\\ \gamma'' = \mathit{upd\_edge}(\gamma', e, v'') \wedge\\ \mathit{compat}(\gamma'', \mathit{finf}, \mathit{tinf}', \mathit{from}, \mathit{to})\end{array}\right\}$
}}}}}
// $\{\mathit{fwd\_postcondition}(\gamma', \gamma'', \mathit{tinf}', \mathit{finf}, \mathit{from}, \mathit{to}, v, n)\} \triangleq \phi_{36}$
```

$$\begin{aligned}\mathit{fwd\_postcondition}(\gamma, \gamma', \mathit{tinf}', \mathit{finf}', \mathit{from}, \mathit{to}, v, n) \triangleq\ & \mathrm{gc\_graph}(\gamma') * \mathrm{finf}(\mathit{finf}') * \mathrm{tinf}(\mathit{tinf}') \wedge\\ & \mathit{compat}(\gamma', \mathit{finf}', \mathit{tinf}', \mathit{from}, \mathit{to}) \wedge \mathit{forward\_relation}(\gamma, \gamma', \mathit{from}, \mathit{to}, v, n)\end{aligned}$$

Fig. 13. Clight code and proof sketch for forward

The function `forward` is the GC's workhorse. When correctly given the spaces `from` and `to` and a pointer `p` to a memory block in `from`, it copies the memory block to the next available location in the `to` space. The function is robust: if passed a "pointer" argument that is actually a data value, or is a pointer that points outside of `from`, it behaves appropriately by taking no action. As we will see in §6.5, these checks are nontrivial. The function is also versatile: it is used to collect the mutator's `args` (which are $*$-separated from the heap) and also to collect the blocks *in* the heap that are reachable via `args`. Its behavior needs to be subtly different in these two cases. Figure 13 shows a decorated proof sketch of `forward` in the latter case, which is harder to verify.

Two abstractions struct thread_inf and file_info—*finf* and *tinf*—represent the graph's metainformation, and together allow us to extract the args array. The proposition *compat* encapsulates a series of checks, *e.g.* to avoid out-of-bounds issues. The arguments s, l, and n are straightforwardly explained on line 1. We are forwarding p, which denotes the *n*th edge of vertex *v*. For readability, we denote the facts known to us in *e.g.* line 1 by ϕ_1, and then use ϕ_1 as a fact in later annotations.

Line 15 shows the case when the block passed to the function was already forwarded. The block's header is zeroed out and its 0^{th} field holds the address of its copy[6], so we simply reroute to the copy. Line 17 shows that this operation gives us a new graph, $\gamma' = \mathit{upd\_edge}(\gamma, e, \mathit{copy}(\gamma, v'))$. That is, in γ, update the edge e to point at $\mathit{copy}(\gamma, v')$. This delicate treatment avoids erroneous double-copying in case a block is reachable from the args array via different paths.

Moving to the meatier case where we must actually make a copy, lines 24 to 26 show how a block is copied over to the next-available spot in the to space. Some of the grungy details having to do with variable-sized memory blocks begin to show up in the C code, but the annotation on line 27 is relatively clean thanks to our mathematical graph framework: this is just the copying of a vertex, and so our new graph after the change is $\gamma' = \mathit{copy\_vertex}(\gamma, to, v', v'')$. Unlike the edit in line 15, which was local to the graph proper, this change spills over to the graph's metainformation: tinf', an alteration to *tinf*, explains that additional space is now used up in the to generation of γ'. The final step (line 32) is to reroute to this new copy, and this is handled exactly as in line 15. The resultant graph is $\gamma'' = \mathit{upd\_edge}(\gamma', e, v'')$. This edit is local to the graph, and so the old metainformation in tinf' remains compatible with the new graph γ''.

The postcondition is a little different from those of find and mark seen earlier: it does not provide a relation saying that forward has acted in a functionally "correct" manner. Rather, it uses *forward_relation* to carefully list the possible end results of calling forward on (v, n) — a vertex may be copied, an edge may be redirected, no action may be taken, *etc.* — and check if the graph γ' falls within one of them. We put the complete definition of *forward_relation*, with its twelve constructors, in the extended online version of our paper. We have such relations for all the key functions in our GC, and our final correctness proof shows that composing these relations together yields the high-level correctness property—a kind of graph isomorphism—that is required by CertiCoq.

6.4 Do Scan

To collect from into to, we first call the function forward_roots, which calls forward on each item in the args array. Thus, no field of args points directly into from. However, the fields of args may still point into from *indirectly*, via the direct links that we just forwarded into to. We fix this via the function do_scan, which scans to and calls forward on all items that were copied over as part of this collection. This fixes any backwards pointers from to into from by moving their targets into to as well. These steps fully disentangle args from the from space, which can be reset to free up memory.

Figure 14 contains a decorated proof sketch of do_scan. The precondition, given on line 1, is very similar to that of forward. We have st and li denoting the boundaries of the from space, and nx denoting the last-used address in the to space. The key difference is that instead of a specific target like p, we take the argument sc, which is the place in the to space from where we must start scanning. Because of the way forward works, we know that the the items recently copied over by forward_roots have been placed placed contiguously between sc and nx. Working from sc upwards, we ignore blocks that are tagged as "do not scan", but otherwise simply call forward on every field of each block. We benefit from forward's robustness: we can trust it to take action only when the field passed to it actually points back into the from space.

[6] These guarantees are set up by forward itself. Refer to lines 28 and 29 of Figure 13 to see this being done straightforwardly.

1 $/\!/\left\{\begin{array}{l}\forall \gamma, \mathit{finf}, \mathit{tinf}, \mathit{from}, \mathit{to}.\\ \mathrm{gc\_graph}(\gamma) * \mathrm{finf}(\mathit{finf}) * \mathrm{tinf}(\mathit{tinf}) \land\\ \mathit{compat}(\gamma, \mathit{finf}, \mathit{tinf}, \mathit{from}, \mathit{to}) \land\\ \mathtt{st} = \mathit{start}(\gamma, \mathit{from}) \land \mathtt{li} = \mathtt{st} + \mathit{gensz}(\gamma, \mathit{from})\\ \land\ \mathtt{sc} = \mathit{scan\_start}(\gamma, \mathit{to}) \land \mathtt{nx} = \mathit{naddr}(\mathit{tinf}, \mathit{to})\end{array}\right\} \triangleq \theta_1$

2 `void do_scan(value *st, *li, *sc, **nx) {`

3 `  value *s; s = sc;`

4 $/\!/\{\theta_1 \land \mathtt{s} = \mathit{scan\_start}(\gamma, \mathit{to})\} \triangleq \theta_4$

5 `  while(s < *nx) {`

6 $/\!/\left\{\begin{array}{l}\theta_4 \land \exists n.\ \mathit{scanned}(\gamma, \mathit{to}) = \mathtt{s} + n \land\\ \mathit{scanned}(\gamma, \mathit{to}) < \mathit{naddr}(\mathit{tinf}, \mathit{to})\end{array}\right\} \triangleq \theta_6$

7 $/\!/\searrow\left\{\begin{array}{l}\exists \mathit{flds}, \mathit{hdr}.\ \mathit{flds} = \mathit{lab}(\mathit{scanned}(\gamma, \mathit{to})) \land\\ \mathit{scanned}(\gamma, \mathit{to}) \mapsto \mathit{flds} \land \mathit{hdr} = \mathit{flds}[-1]\end{array}\right\} \triangleq \theta_7$

8 `    header_t hd = *((header_t *) s);`

9 $/\!/\swarrow\{\theta_7 \land \mathtt{hd} = \mathit{val}(\mathit{hdr})\}$

10 $/\!/\{\theta_6 \land \mathtt{hd} = \mathit{val}(\mathit{hdr})\} \triangleq \theta_{10}$

11 `    mlsize_t sz = Wosize_hd(hd);`

12 $/\!/\{\theta_{10} \land \mathtt{sz} = \mathit{blocksize}(\mathtt{hd})\} \triangleq \theta_{12}$

13 `    int tag = Tag_hd(hd);`

14 `    if (!No_scan(tag)) {`

15 `      intnat j;`

16 `      for(j = 1; j <= sz; j++) {`

17 $/\!/\{\mathit{entails}\ \phi_1\ \ \mathit{where}\ \ v = \mathtt{s}, n = \mathtt{j}\}$

18 `        forward (st, li, nx, &Field(s, j));`

19 $/\!/\left\{\begin{array}{l}\mathit{entails}\ \phi_{36}\ \ \mathit{where}\ \ v = \mathtt{s}, n = \mathtt{j}\\ \mathit{i.e.},\ \exists \gamma', \mathit{tinf}'.\ \mathrm{gc\_graph}(\gamma') \land\\ \mathit{compat}(\gamma', \mathit{finf}, \mathit{tinf}', \mathit{from}, \mathit{to}) \land\\ \mathit{forward\_relation}(\gamma, \gamma', \mathit{from}, \mathit{to}, \mathtt{s}, \mathtt{j})\end{array}\right\} \triangleq \theta_{19}$

20 `      }`

21 `    }`

22 `    s += 1+sz;`

23 `  }}`

24 $/\!/\{\mathit{ds\_postcondition}(\gamma', \gamma', \mathit{tinf}', \mathit{finf}, \mathit{from}, \mathit{to})\}$

$$\begin{aligned}\mathit{ds\_postcondition}(\gamma, \gamma', \mathit{tinf}', \mathit{finf}', \mathit{from}, \mathit{to}) \triangleq\ & \mathrm{gc\_graph}(\gamma') * \mathrm{finf}(\mathit{finf}') * \mathrm{tinf}(\mathit{tinf}') \land\\ & \mathit{compat}(\gamma', \mathit{finf}', \mathit{tinf}', \mathit{from}, \mathit{to}) \land \mathit{do\_scan\_relation}(\gamma, \gamma', \mathit{from}, \mathit{to})\end{aligned}$$

Fig. 14. Clight code and proof sketch for do_scan

The only use of LOCALIZE is when reading the block header on line 8, and this is not very different from the read seen in `forward`. Lines 17 and 19 represent exactly the pre- and postconditions of `forward` for some vertex (s, j). After the for-loop, the entire block represented by s has either been forwarded (line 20) or ignored (line 21). This may increment `nx`, so we continue until `sc = nx`.

At first glance, this verification may seem quite elementary. Its trickiness comes from the fact that `do_scan` operates exactly on the interface between the Clight code and the mathematical graph model introduced in §6.2. Functions like `forward` ignore the Clight memory representation of the heap and go about their business in the abstract domain of mathematical graphs, but `do_scan` cannot do this because grungy details such as the index order of runtime-allocated blocks are key to its strategy of a linear search from `sc` to `nx`.

6.5 Performance and Overflows and Undefined Behaviours, Oh My!

Bugs in the GC code. We discovered and fixed two bugs in the source code during our verification. The first was a performance bug we discovered when developing the key invariants. The original GC code executed Cheney's algorithm too conservatively, scanning the entire `to` space for backward pointers into `from`. We showed that scanning a subset of `to` suffices. Performance doubled. The second bug was an overflow when subtracting two pointers to calculate the size of a space, as below. Here the pointers `start` and `limit` point to the beginning and end of the `i`th space of the heap `h`.

```
int w = h->spaces[i].limit - h->spaces[i].start;
```

This subtraction is defined in C and Clight, but overflows if the difference equals or exceeds 2^{31}. We adjusted the size of the largest generation to avoid this overflow.

Undefined behavior in C. We found two places where the semantics of Clight is unable to specify an OCaml-style GC such as ours. The first area of undefined behavior results from our GC's use of the well-established 31-bit integer trick to allow both boxed and unboxed data in block fields [Hickey et al. 2014]. To distinguish them, `forward` calls `Is_block` (line 5), which in turn calls the following:

```
int test_int_or_ptr (value x) { return (int)(((intnat)x)&1); }
```

This function aims to return 1 if `x` is an int, and 0 if it is an aligned pointer. When `x` is an integer, this is indeed well-defined. But when `x` is a pointer, the code gets stuck because taking the logical `and` of a pointer is undefined in Clight. The second issue involves double-bounded pointer comparisons. On line 6 of Figure 13, `forward` checks whether the object it is considering, which it already knows to be a pointer, is in fact pointing into the `from` space. The `forward` function uses the following:

```
int Is_from(value * from_start, value * from_limit, value * v) {
  return (from_start <= v && v < from_limit); }
```

Here, the `start` and `limit` pointers are in the same memory block. If `v` is also in the same block, `Is_from` correctly computes whether it is in bounds. However, if `v` is in a different block, the comparison gets stuck—both halves of the conjunction are undefined—instead of returning `false`.

Although the Clight code is undefined, we used CompCert's "extcall_properties" to prove that CompCert's compiler transformations will preserve the necessary invariants for both operations. Both operations require careful treatment. The first operation requires that the `x` pointer is defined and has even offset within its memory block, *and* that `x+1` is also defined. This length-two requirement ensures that `x` is not pointing to a stack-allocated local variable of type `char`, which CompCert can realign as it assembles the stack frame. The second operation is respected by CompCert because the pointer comparison is double-bounded (below by `from_start` and above by `from_limit`); in contrast, a single-bounded comparison need not be semantically preserved by CompCert.

Other than these two items, the GC is fully defined in Clight—we were even able to prove that all casts (*e.g.* line 8) are well-defined. As a concluding thought, Coq itself is written in OCaml with a similar-style garbage collector. Thus, our GC is at least as well-defined as Coq itself.

7 ENGINEERING OUR TECHNIQUES

An important feature of our work is that it integrates into substantial existing projects, making our techniques open to a large userbase. Another feature is that we carefully organized our library to separate the concerns of abstract mathematical reasoning and code-specific spatial reasoning. §7.1 explains the former effort, and §7.2 the latter. §7.3 gives statistics about our development.

7.1 Localizations in VST with `localize` and `unlocalize`

CompCert is a fully machine-checked verified compiler for CompCert C [Leroy 2006]. The Verified Software Toolchain consists of a series of machine-checked modules written in Coq to reason about CompCert C programs [Appel et al. 2014]. Floyd is VST's module for verifying such programs using separation logic. VST's modules interlock so there are no "gaps" in the end-to-end certified results; accordingly all of the rules employed by the Floyd module have been proved sound with respect to the underlying semantics used by CompCert. Floyd is written in Ltac and Gallina and is designed to help users verify the full functional correctness of their programs. We added two tactics, `localize` and `unlocalize`, to integrate the Localize rule (as described in §2–§3) into Floyd.

The Floyd module presents users with a pleasant "decorated program" visualization for Hoare proofs, in which users work from the top of the program to the bottom even though the formal proof is maintained as applications of inference rules. For example, suppose the proof goal is $\{P_1\}\, c_1 ; c_2\, \{P_5\}$ and VST's user tells Floyd to apply a Hoare rule for c_1, *e.g.* $\{P_1\}\, c_1\, \{P_2\}$. Floyd then automatically applies the Sequence rule and show the user $\{P_2\}\, c_2\, \{P_5\}$ as the remaining goal. When the user is in the middle of a verification, the decorated program is partially done (*i.e.* the

1	{ P_1 }	{ P_1 }	{ P_1 }	{ P_1 }
2	c1	c1	c1	c1
3	{ P_2 }	{ P_2 }	{ P_2 }	{ P_2 }
4	↘ { P_3 }	{ $?F * P_3$ }	↘ { P_3 }	{ $?F * P_3$ }
5	c2;	c2;	c2;	c2;
6	{ P_4 }	{ $?F * P_4$ }	{ P_4 }	{ $?F * P_4$ }
7			c3;	c3;
8	. . .	. . .	↙ { P_5 }	{ $?F * P_5$ }
9			{ P_6 }	{ P_6 }
10			. . .	. . .
	(front)	(back)	(front)	(back)

Fig. 15. Front and back ends of `localize` and `unlocalize`

proof is finished from the top to "the current program point") and the inference tree is also partially done (*i.e.* with holes that are represented by the remaining proof goals in Coq).

We wish to preserve this "decorated program" view while extending Floyd to support localization. Our task therefore is to construct a proof in Coq's underlying logic that allows a localization block to be constructed in this manner—that is, we wish to enter a localization block without requiring the user to specify the "exit point" in advance. The engineering is tricky because the proof Floyd is constructing (*i.e.* applications of inference rules) has holes in places where the user's "top to bottom" view of things has not yet arrived.

Figure 15 has two partially-decorated "proofs in progress", from both the user's (front end) and Floyd's (back end) points of view. In the first column, from the user's point of view, they see the assertion P_2 (line 3) and decide to use the `localize` tactic to zoom into P_3 (line 4). They then apply some proof rules to move past c_2 to reach the assertion P_4 (line 6). At this point, Floyd does not know when the corresponding `unlocalize` tactic will execute, so it does not know which commands will be inside the block or what the final local and global postconditions will be.

Accordingly, the `localize` tactic builds an incremental proof in the underlying program logic by applying Frame with an uninstantiated metavariable. The second column of Figure 15 shows the back end with the unknown frame $?F$, which will eventually be instantiated by `unlocalize`.

In the third column, the user has advanced past c_3 to reach the local postcondition P_5 and now wishes to `unlocalize` to P_6. Afterwards, the internal state looks like the fourth column, and so to a first approximation, `unlocalize` can instantiate $?F$ with $P_5 \mathrel{-\!\!*} P_6$. In truth, $?F$ is chosen more subtly to properly handle both existential variables and modified program variables. Then `unlocalize` automatically simplifies the goals to present a cleaner interface to the user. These transformations require the additional theory given in §3.

7.2 Modularity of our Library

We worked to make our library modular, thus encouraging proof reuse. Figure 16 gives a sense of this engineering effort with the garbage collector as a case study. 14 files were used to verify the GC, and the bar at the bottom the relative lengths of those files on a linear scale. Files 1 and 3 are about half the development, and they contain generic definitions and theorems. They can be reused for the verification of programs that have mathematical or spatial structures similar to our GC.

Moving to the circular graph, each straight line perpendicular to the perimeter represents a theorem used to verify the GC. The lengths of those lines correspond to the lengths of the relevant theorems on a logarithmic scale. The 14 color-coded sectors represent the files in which these theorems are housed. Inside the circle, arcs connecting theorems represent dependencies, where an arc is colored the same as the theorem's *caller*.

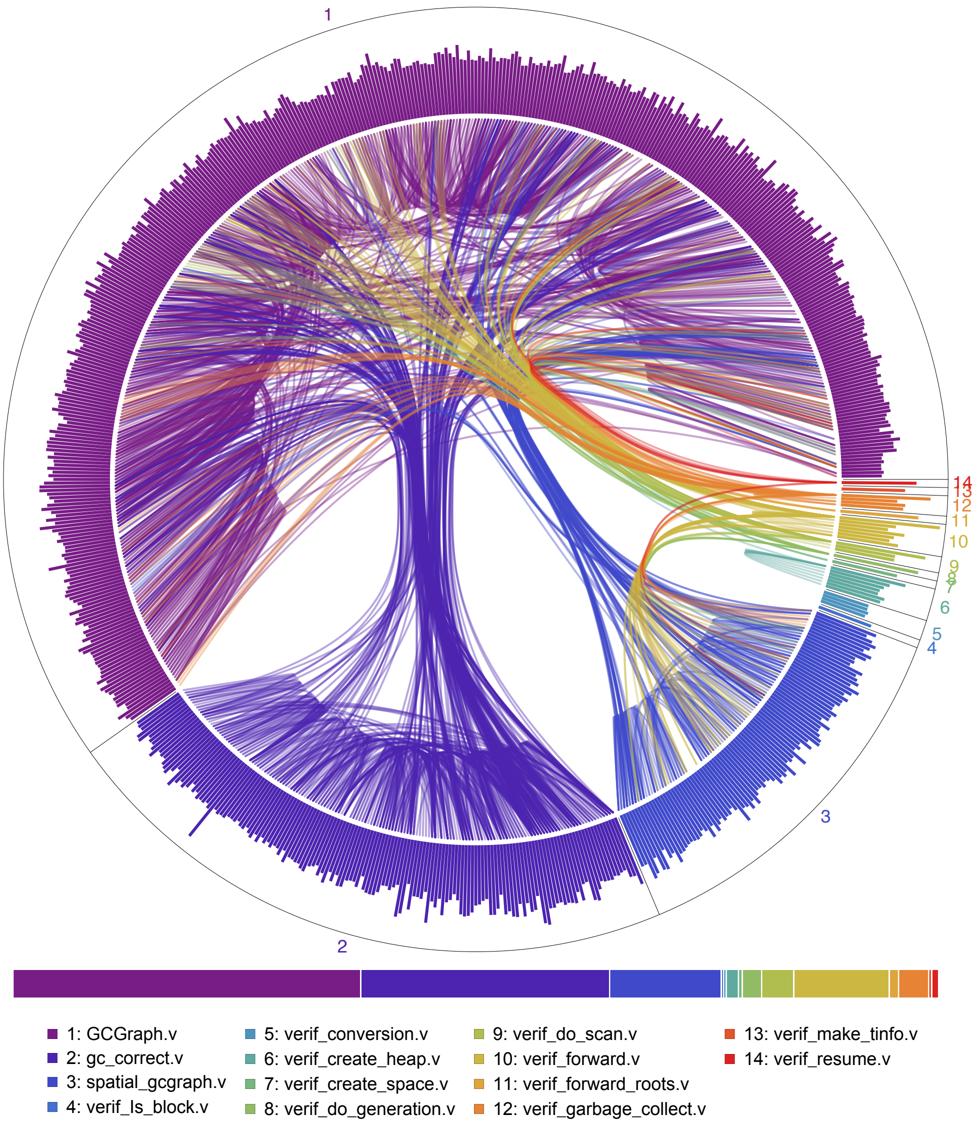

Fig. 16. Theorems in the verification of the GC

Sector 1 contains 432 generic helper theorems used to check that functions satisfy the inductive spatial relations that we need. These theorems do not depend on any other GC theorems. Sector 3 contains 91 theorems that establish spatial correctness (see §5). Sectors 4-14 are the actual proof scripts for individual functions of the C code. Unsurprisingly, they depend chiefly on Sectors 1 and 3. Among these, sector 10 is notably large, and this makes sense given that it houses our workhorse `forward` function. The C code of `forward` is 35 lines, and its verification is 1309 lines long: a 37-fold blowup. Sector 2 contains 155 theorems that prove the final mathematical graph isomorphism (see §4). These theorems depend only on Sector 1, and no other sectors depend on them. This is to be expected, seeing as graph isomorphism is our final goal.

Table 1. Statistics for our code base

Component	Section	Files	Size (in lines)	Definitions	Theorems
Common Utilities		10	3,578	44	289
Math Graph Library	§4	20	10,585	216	581
Spatial Graph Library	§5	3	2,328	59	110
Integration into VST	§3,§7	11	2,783	17	172
Marking (graph and DAG)	§3	6	775	9	20
Spanning Tree	§3	5	2,723	17	92
Union-Find (heap and array)	§2	18	3,193	107	135
Garbage Collector	§6	16	13,858	235	712
Total Development		89	39,823	704	2,111

7.3 Statistics Related to our Development

All our results in this paper have been machine-checked. Although the size of a development does not perfectly match its importance or the difficulty of its implementation, we present the size nonetheless in Table 1. Our proofs are written in a very dense style. For comparison, verifying a simple 39-line list-based merge sort in VST takes 600 lines. At ≈ 400 LOC, the garbage collector is much larger, and is very complicated both mathematically and spatially, in many places teetering on the edge of what can be defined in C. CompCert has 217k LOC, 5,687 definitions, and 6,694 theorems; VST has 623k LOC, 14,038 definitions, and 21,442 theorems. It is hard to determine the time taken for this project on the whole, but the verification of the garbage collector took 8 months.

8 RELATED WORK

Comparison with Hobor and Villard [2013]. Our work builds on the theory of ramification by Hobor and Villard, who verified graph algorithms on pen-and-paper using their Ramify rule:

$$\textsc{Ramify}\quad \frac{\{L_1\}\ c\ \{L_2\} \qquad G_1 \vdash L_1 * (L_2 -\!\!* \ G_2)}{\{G_1\}\ c\ \{G_2\}} \qquad \mathit{freevars}(L_2 -\!\!* \ G_2) \cap \mathsf{ModVar}(c) = \varnothing$$

Our Localize rule upgrades Ramify to better handle modified program variables (note the side condition and recall the discussion in §3) and existential quantifiers in postconditions. Hobor and Villard avoided these challenges by proposing a unwieldy variant of Ramify called RamifyAssign, which could reason about the special case of a single assignment `x=`$f(\ldots)$, assuming the verifier can make the local program translation to `x'=`$f(\ldots)$`; x=x'`, where `x'` is fresh. This is nontrivial in large existing formal developments, such as VST, that do not have any way to prove programs equivalent. Hobor and Villard could not verify unmodified program code, modify program variables inside nested localization blocks, or handle multiple assignments in a single block as in lines 15–17 of Figure 3. They avoided existentials in localized postconditions by defining all mathematical operations (*e.g. mark*) as functions rather than as relations; this is fine for pen-and-paper, but painful in a mechanized setting wherein functions must be proven to terminate.

Hobor and Villard treated mathematical graphs as triples (V, E, L) of vertices, edges, and a vertex labeling function, where vertices had no more than two neighbors. Our mathematical graph framework (§4) is more modular and versatile, and ships with hundreds of reusable definitions and theorems. Further, our library has been tuned to work smoothly in a mechanized context.

Hobor and Villard erroneously defined spatial graphs recursively. Unfortunately, other members of the research community (*e.g.* Raad et al. [2015]) followed their lead. We expose this error (§5.2) and provide a sound and rather general definition for graph that recovers fold/unfold reasoning (§5.3). We develop a much more general and more modular set of related lemmas and connected our spatial reasoning to the verification framework of CompCert/VST (§7). Our development is entirely machine-checked (§7) whereas they used only pen and paper.

Other Pen-and-Paper Verification of Graph Algorithms and/or $\cup\!\!\!\!*$. Yang [2001] verified the Schorr-Waite algorithm, and this is widely considered a landmark in the early separation logic literature. Bornat et al. [2004] gave an early attempt to reason about graph algorithms in separation logic in a more general way. Krishnaswami [2011] provided the first separation logic proof of union-find.

Reynolds [2003] was the first to document the overlapping conjunction $\cup\!\!\!\!*$, albeit without any strategy to reason about it using Hoare rules. Gardner et al. [2012] were the first to reason about a program using $\cup\!\!\!\!*$ in Javascript. Raad et al. [2015] used $\cup\!\!\!\!*$ within their CoLoSL program logic to reason about a concurrent spanning algorithm using a kind of “concurrent localization”.

Machine-Checked Verification of Graph Algorithms. A decade after Yang verified Schorr-Waite on paper, Leino [2010] automated its verification in Dafny. Sergey et al. [2015] verified a concurrent spanning tree algorithm, and moreover developed mechanized Coq proofs. Their algorithm was written in FCSL, a monadic DSL that combines effectful operations with pure Coq expressions; FSCL cannot be executed. Chen et al. [2018] compared how three provers (Coq, Isabelle, and Why3) can verify Tarjan’s strongly-connected component algorithm written in the native language of each of the tools. Because these are written in the native languages of a proof assistant, they avoid “real-world” language concerns such as memory models and overflow.

Lammich and Neumann [2015] extended the Isabelle Refinement Framework to verify a range of DFS algorithms via stepwise refinement. Their framework allows the reuse of previously-proved DFS invariants by establishing an inductive “most specific invariant” and deriving other inductive invariants from it. Lammich and Sefidgar [2019] extended this further and presented verifications of the correctness and time complexity of the Edmonds-Karp and push-relabel algorithms. Lammich et al. produced very readable proofs of classic textbook algorithms by using the Isar language atop of their Isabelle proofs. They used Isabelle’s code generator to export efficient executable code, but with the caveat that the code comes with a guarantee of only partial correctness semantics.

Charguéraud [2011] used his CFML tool to Coq-verify an OCaml implementation of Dijkstra. Guéneau et al. [2019] extended CFML and verified the correctness and time complexity of a modified version of the BFGT cycle-detection algorithm. The graph algorithms verified in CFML tend to be “graph theory” in flavour, whereas the algorithms we have verified tend to have more of a “systems” flavor. This difference is partially explained by the fact that code written in ML can take advantage of its high-level design, whereas code written in C is often interested in handling grungy systems tasks. For example, references in ML cannot be null and do not support pointer arithmetic; of course both are possible—and lead to nontrivial complications—in C. Accordingly, the CFML proofs benefit from ML’s cleaner computational model. Our verifications are in C so we must contend with C’s memory model, pointer arithmetic, significant scope for undefined behavior, and so forth.

Charguéraud and Pottier [2019] used CFML to verify the correctness and time complexity of union-find. Their work is an interesting counterpoint to ours because, while it maintains an abstraction between the client and the internal mathematical/spatial facts that the client need not know, it does not maintain a separation between the mathematical and spatial facts themselves, as we do in §4 and §5. This separation is worthwhile: our modular method let us verify an alternate version of union-find that uses an array of vertices rather than individually heap-allocated nodes. This secondary verification then used *exactly the same* mathematical proof of functional correctness despite the radically different layout of spatial memory. Our work does not verify the time complexity of union-find. When we attempted to prove the necessary amortisation bounds we ran into an overflow issue: it was impossible to prove that the rank would not exceed max_int because the CompCert memory model does not place a bound on the total number of allocations. Informally, this overflow is impossible in practice because no computer has $2^{2^{64}}$ bytes of memory, which would be required for this overflow to occur, but Coq remains unconvinced. Charguéraud and Pottier acknowledged and sidestepped this issue by representing rank using the Coq type Z, which was not an option for us given the end-to-end nature of the VST+CompCert toolchain.

Verification Tools in Coq. Our work interacts with the Floyd verification module within the Verified Software Toolchain (VST) [Appel et al. 2014]. The Floyd module uses tactics to enable the separation-logic verification of CompCert C programs. VST connects to the CompCert certified C compiler [Leroy 2006], and thus has no gaps or admits between the verified source code and the eventual assembly code [Appel 2012].

Charge! likewise uses Coq tactics to work with a shallow embedding of higher order separation logic, but focuses on OO programs written in Java/C# [Bengtson et al. 2012]. Iris Proof Mode provides a similar framework for higher-order concurrent reasoning in Coq [Krebbers et al. 2017].

CFML enables the verification of OCaml programs by reasoning about their "characteristic formulae" in separation logic using Coq [Charguéraud 2010, 2011]. CFML has been used to verify a range of functional and imperative programs, including some graph-related algorithms as discussed above. Charguéraud and Pottier [2019] extended CFML to reason about time credits. The work of Guéneau et al. [2017] indicates that CFML is exploring a connection with the certified CakeML compiler [Kumar et al. 2014].

While the tools above require substantial human guidance, Bedrock [Chlipala 2011] is a more automated approach to the verification of low level programs using separation logic in Coq. Bedrock leverages the fact that phrasing function specifications in a *computational* style (in this case, inspired by functional programming) leads to separation logic proof obligations that are quite automatable. It simplifies these obligations into pure mathematics using a custom workhorse tactic, and then discharges those obligations using standard Coq automation.

Other Verification Tools. Many more-automated verification tools also use separation logic in a forward reasoning style. Smallfoot [Berdine et al. 2005], jStar [Distefano and Parkinson 2008], HIP/SLEEK [Chin et al. 2010], and Verifast [Jacobs et al. 2011] are landmarks at various points on the expressibility-automatability spectrum. KeY [Beckert et al. 2007] and Dafny [Leino 2010] are verifiers that are not based on separation logic. KeY uses an interactive verifier while Dafny pursues automation with Z3 [de Moura and Bjørner 2008].

Mechanized Mathematical Graph Theory. There is a long history, going back at least 28 years, of mechanized reasoning about mathematical graphs [Wong 1991]. The most famous mechanically verified "graph theorem" is the Four Color Theorem [Gonthier 2005]; however the development actually uses hypermaps instead of graphs. In general most "mathematical graph" frameworks in the literature [Butler and Sjogren 1998; Chou 1994; Duprat 2001; Nipkow 2016; Nordhoff and Lammich 2012; Ridge 2005; Tamai 2000; Wong 1991; Yamamoto et al. 1995, 1998] were not used to verify real code, for which they seem unsuitable. Verifying real code requires delicate concepts such as removing a subgraph, null nodes, and parallel edges, and one of our contributions is that our framework is general enough to support such verification. Noschinski [2015b] built a graph library in Isabelle/HOL whose formalization is the closest to ours, *e.g.* supporting graphs with labeled and parallel arcs. Beyond being in Coq, our setup supports at least three features beyond Noschinski's: reasoning about incomplete graphs (as discussed in §4.1 using figure 5), labeling the graph as a whole (used, for example, in the garbage collector to store metainformation about the number and location of the generations), and our modular typeclass-supported "graphs with properties" setup in General Graph (as described in §4.2). Dubois et al. [2015] and Noschinski [2015a] used proof assistants to design verifiable checkers for solutions to graph problems. Bauer and Nipkow [2002] and Yamamoto et al. [1995] used an inductive encoding of graphs to formalize planar graph theory.

Verification of Garbage Collection Algorithms. Schism [Gammie et al. 2015; Pizlo et al. 2010] is a certified concurrent collector built in a Java VM that services multi-core architectures with weak memory consistency. McCreight et al. [2010, 2007] introduced GCminor, which is a certified translation step added to CompCert's translation from Clight to assembly. GCminor makes explicit the specific invariants that the garbage collector relies upon, thus minimising errors due to the violation of invariants between the garbage collector and the mutator. Petrank and Hawblitzel [2010] annotated x86 code for two GCs by hand, and then used Boogie and the Z3 automated theorem prover to verify their correctness automatically.

The closest piece of work to our certified GC is probably the excellent certified GC for the Cake ML project [Ericsson et al. 2017], since both integrate a certified GC into a certified compiler for a functional language. Their GC is written closer to assembly than C, which is both a positive—in that they avoid undefined behaviors—and a negative, in that their GC is harder to understand and upgrade and cannot take advantage of the mature CompCert compiler. Their GC lacks some of our optimisations (*e.g.* they have only three generations), but on the other hand handles mutation in the GC heap. The largest difference, however, is that we present an integrated graph framework suitable for reasoning about many graph algorithms, of which our GC is merely the flagship. In contrast, they focus much more narrowly on the problem of certified GCs.

9 FUTURE WORK AND CONCLUSION

In the future we plan to improve the pure reasoning of graphs and similar data structures, with a particular focus on automation. We have also begun to investigate integrating our techniques into the HIP/SLEEK toolchain [Chin et al. 2010], which, as compared to VST, provides more automation at the cost of lower expressivity. We are also interested in investigating better ways to handle the kinds of undefined behavior upon which real C systems code sometimes relies.

Our main contributions were as follows. We developed a mathematical graph library that was powerful enough to reason about graph-manipulating algorithms written in real C code. We connected these mathematical graphs to spatial graphs in the heap via separation logic. We developed localization blocks to smoothly reason about a local action's effect on a global context in a mechanized context, including a robust treatment of modified program variables and existential quantifiers in postconditions. We demonstrated our techniques on several nontrivial examples, including union-find and spanning tree. Our flagship example was the verification of the garbage collector for the CertiCoq project, during which we found two places in which the C semantics is too weak to define an OCaml-style GC. We integrated our techniques into the VST toolset.

ACKNOWLEDGMENTS

We thank Asankhaya Sharma for his help with a previous version of this paper, Neel Krishnaswami for his helpful suggestions and encouragements, and Xavier Leroy and Robbert Krebbers for fruitful discussions. We also thank the CertiCoq team (esp. Andrew W. Appel, Olivier Savary Belanger, and Zoe Paraskevopoulou) for their overall support and for hosting Shengyi Wang for a summer. This work was funded in part by the Yale-NUS College grant R-607-265-322-121, the National Science Foundation grant CCF-1521602, and the Shanghai Pujiang Program grant 19PJ1406000. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of Yale-NUS College, the National Science Foundation, or the Shanghai Pujiang Program.

REFERENCES

Abhishek Anand, Andrew Appel, Greg Morrisett, Zoe Paraskevopoulou, Randy Pollack, Olivier Savary Belanger, Matthieu Sozeau, and Matthew Weaver. 2017. CertiCoq: A verified compiler for Coq. In *The Third International Workshop on Coq for Programming Languages (CoqPL)*.

Andrew W. Appel. 2012. Verified Software Toolchain. In *NASA Formal Methods - 4th International Symposium, NFM 2012, Norfolk, VA, USA, April 3-5, 2012. Proceedings*. 2. https://doi.org/10.1007/978-3-642-28891-3_2

Andrew W. Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. 2014. *Program Logics for Certified Compilers*. Cambridge University Press, New York, NY, USA.

Andrew W. Appel and David McAllester. 2001. An Indexed Model of Recursive Types for Foundational Proof-Carrying Code. *ACM Transactions on Programming Languages and Systems* 23(5) (2001), 657–683.

Andrew W. Appel, Paul-André Melliès, Christopher D. Richards, and Jerôme Vouillon. 2007. A Very Modal Model of a Modern, Major, General Type System. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'07)*. 109–122.

Gertrud Bauer and Tobias Nipkow. 2002. The 5 colour theorem in Isabelle/Isar. In *International Conference on Theorem Proving in Higher Order Logics*. Springer, 67–82.

Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt. 2007. *Verification of Object-oriented Software: The KeY Approach*. Springer-Verlag, Berlin, Heidelberg.

Jesper Bengtson, Jonas Braband Jensen, and Lars Birkedal. 2012. Charge! - A Framework for Higher-Order Separation Logic in Coq. In *Interactive Theorem Proving - Third International Conference, ITP 2012, Princeton, NJ, USA, August 13-15, 2012. Proceedings*. 315–331.

Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. 2005. Smallfoot: Modular Automatic Assertion Checking with Separation Logic. In *FMCO*. 115–137.

R. Bornat, C. Calcagno, and P. O'Hearn. 2004. Local reasoning, separation and aliasing. In *SPACE*, Vol. 4.

Richard Bornat, Cristiano Calcagno, and Hongseok Yang. 2006. Variables as Resource in Separation Logic. *ENTCS* 155 (2006), 247–276.

Ricky W Butler and Jon A Sjogren. 1998. *A PVS Graph Theory Library*. Technical Report.

Arthur Charguéraud. 2010. Program verification through characteristic formulae. In *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*. 321–332. https://doi.org/10.1145/1863543.1863590

Arthur Charguéraud. 2011. Characteristic formulae for the verification of imperative programs. In *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*. 418–430. https://doi.org/10.1145/2034773.2034828

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, 3 (2019), 331–365. https://doi.org/10.1007/s10817-017-9431-7

Ran Chen, Cyril Cohen, Jean-Jacques Lévy, Stephan Merz, and Laurent Théry. 2018. Formal Proofs of Tarjan's Algorithm in Why3, Coq, and Isabelle. *CoRR* abs/1810.11979 (2018). arXiv:1810.11979 http://arxiv.org/abs/1810.11979

C. J. Cheney. 1970. A nonrecursive list compacting algorithm. *Commun. ACM* 13, 11 (1970), 677–678. https://doi.org/10.1145/362790.362798

Wei Ngan Chin, Cristina David, Huu Hai Nguyen, and Shengchao Qin. 2010. Automated verification of shape, size and bag properties via user-defined predicates in separation logic. *Science of Computer Programming* 77(9) (2010), 1,006–1,036.

Adam Chlipala. 2011. Mostly-automated verification of low-level programs in computational separation logic. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*. 234–245.

Ching-Tsun Chou. 1994. A Formal Theory of Undirected Graphs in Higher-Order Logic. In *Higher Order Logic Theorem Proving and Its Applications*. Springer, 144–157.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford S. Stein. 2009. *Introduction to algorithms, 3rd edition*. MIT Press and McGraw-Hill.

Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*. 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

Dino Distefano and Matthew J. Parkinson. 2008. jStar: towards practical verification for java. In *Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2008, October 19-23, 2008, Nashville, TN, USA*. 213–226. https://doi.org/10.1145/1449764.1449782

Robert Dockins, Aquinas Hobor, and Andrew W. Appel. 2009. A Fresh Look at Separation Algebras and Share Accounting. In *Programming Languages and Systems, 7th Asian Symposium, APLAS 2009, Seoul, Korea, December 14-16, 2009. Proceedings*.

161–177.

Catherine Dubois, Sourour Elloumi, Benoit Robillard, and Clément Vincent. 2015. Graphes et couplages en Coq. In *Vingt-sixièmes Journées Francophones des Langages Applicatifs (JFLA 2015)*.

Jean Duprat. 2001. A Coq toolkit for graph theory. *Rapport de recherche* 15 (2001).

Adam Sandberg Ericsson, Magnus O. Myreen, and Johannes Åman Pohjola. 2017. A Verified Generational Garbage Collector for CakeML. In *Interactive Theorem Proving - 8th International Conference, ITP 2017, Brasília, Brazil, September 26-29, 2017, Proceedings*. 444–461. https://doi.org/10.1007/978-3-319-66107-0_28

Peter Gammie, Antony L Hosking, and Kai Engelhardt. 2015. Relaxing safely: verified on-the-fly garbage collection for x86-TSO. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 99–109.

Philippa Gardner, Sergio Maffeis, and Gareth David Smith. 2012. Towards a program logic for JavaScript. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*. 31–44. https://doi.org/10.1145/2103656.2103663

Georges Gonthier. 2005. A computer-checked proof of the four colour theorem.

Armaël Guéneau, Jacques-Henri Jourdan, Arthur Charguéraud, and François Pottier. 2019. Formal proof and analysis of an incremental cycle detection algorithm. In *Interactive Theorem Proving - 9th International Conference, ITP 2019, Portland, USA, September 8-13, 2019, Proceedings*.

Armaël Guéneau, Magnus O. Myreen, Ramana Kumar, and Michael Norrish. 2017. Verified Characteristic Formulae for CakeML. In *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*. 584–610. https://doi.org/10.1007/978-3-662-54434-1_22

Jason Hickey, Anil Madhavapeddy, and Yaron Minsky. 2014. *Real World OCaml*. OReilly.

Aquinas Hobor and Jules Villard. 2013. The ramifications of sharing in data structures. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'13)*. 523–536.

Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. 2011. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*. 41–55.

Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017. Interactive proofs in higher-order concurrent separation logic. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. 205–217. http://dl.acm.org/citation.cfm?id=3009855

Neelakantan R. Krishnaswami. 2011. *Verifying Higher-Order Imperative Programs with Higher-Order Separation Logic*. Ph.D. Dissertation.

Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: a verified implementation of ML. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*. 179–192. https://doi.org/10.1145/2535838.2535841

Peter Lammich and René Neumann. 2015. A Framework for Verifying Depth-First Search Algorithms. In *Proceedings of the 2015 Conference on Certified Programs and Proofs, CPP 2015, Mumbai, India, January 15-17, 2015*. 137–146. https://doi.org/10.1145/2676724.2693165

Peter Lammich and S. Reza Sefidgar. 2019. Formalizing Network Flow Algorithms: A Refinement Approach in Isabelle/HOL. *J. Autom. Reasoning* 62, 2 (2019), 261–280. https://doi.org/10.1007/s10817-017-9442-4

K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers*. 348–370. https://doi.org/10.1007/978-3-642-17511-4_20

Xavier Leroy. 2006. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*. 42–54.

Andrew McCreight, Tim Chevalier, and Andrew Tolmach. 2010. A certified framework for compiling and executing garbage-collected languages. In *ACM Sigplan Notices*, Vol. 45. ACM, 273–284.

Andrew McCreight, Zhong Shao, Chunxiao Lin, and Long Li. 2007. A general framework for certifying garbage collectors and their mutators. In *ACM SIGPLAN Notices*, Vol. 42. ACM, 468–479.

Tobias Nipkow. 2016. Verified analysis of functional data structures. In *1st International Conference on Formal Structures for Computation and Deduction (FSCD 2016)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

Benedikt Nordhoff and Peter Lammich. 2012. Dijkstra's Shortest Path Algorithm. *Archive of Formal Proofs* (Jan. 2012). http://isa-afp.org/entries/Dijkstra_Shortest_Path.shtml, Formal proof development.

Lars Noschinski. 2015a. *Formalizing Graph Theory and Planarity Certificates*. Ph.D. Dissertation. Universität München.

Lars Noschinski. 2015b. A Graph Library for Isabelle. *Mathematics in Computer Science* 9, 1 (2015), 23–39. https://doi.org/10.1007/s11786-014-0183-z

Peter O'Hearn, John Reynolds, and Hongseok Yang. 2001. Local Reasoning about Programs that Alter Data Structures. In *Computer Science Logic*, Laurent Fribourg (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–19.

Peter W. O'Hearn. 2012. A Primer on Separation Logic (and Automatic Program Verification and Analysis). *Software Safety and Security* 33 (2012), 286–318.

Erez Petrank and Chris Hawblitzel. 2010. Automated Verification of Practical Garbage Collectors. *Logical Methods in Computer Science* 6 (2010).

Filip Pizlo, Lukasz Ziarek, Petr Maj, Antony L Hosking, Ethan Blanton, and Jan Vitek. 2010. Schism: fragmentation-tolerant real-time garbage collection. *ACM Sigplan Notices* 45, 6 (2010), 146–159.

Azalea Raad, Jules Villard, and Philippa Gardner. 2015. CoLoSL: Concurrent Local Subjective Logic. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings.* 710–735. https://doi.org/10.1007/978-3-662-46669-8_29

John C. Reynolds. 2003. A Short Course on Separation Logic. (2003). http://www.cs.cmu.edu/afs/cs.cmu.edu/project/fox-19/member/jcr/wwwaac2003/notes7.ps.

Tom Ridge. 2005. Graphs and Trees in Isabelle/HOL. (2005).

Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. 2015. Mechanized Verification of Fine-grained Concurrent Programs. In *PLDI*. 77–87.

Tetsuo Tamai. 2000. Formal treatment of a family of fixed-point problems on graphs by CafeOBJ. In *Formal Engineering Methods, 2000. ICFEM 2000. Third IEEE International Conference on*. IEEE, 67–74.

Alfred Tarski. 1955. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics* 5 (1955), 285–309.

Shengyi Wang, Qinxiang Cao, Anshuman Mohan, and Aquinas Hobor. 2019. Extended Autoquack. https://www.comp.nus.edu.sg/~hobor/Publications/2019/autoquack_extended_oopsla19.pdf

Wai Wong. 1991. A Simple Graph Theory And Its Application In Railway Signaling. In *HOL Theorem Proving System and Its Applications, 1991., International Workshop on the*. 395–409. https://doi.org/10.1109/HOL.1991.596304

Mitsuharu Yamamoto, Shin-ya Nishizaki, Masami Hagiya, and Yozo Toda. 1995. Formalization of planar graphs. In *International Conference on Theorem Proving in Higher Order Logics*. Springer, 369–384.

Mitsuharu Yamamoto, Koichi Takahashi, Masami Hagiya, Shin-ya Nishizaki, and Tetsuo Tamai. 1998. Formalization of graph search algorithms and its applications. In *International Conference on Theorem Proving in Higher Order Logics*. Springer, 479–496.

Hongseok Yang. 2001. *Local Reasoning for Stateful Programs.* Ph.D. Dissertation. University of Illinois.