Certifying Graph-Manipulating C Programs via Localizations within Data Structures

We develop powerful and general techniques to mechanically verify realistic programs that manipulate heap-represented graphs~~and related data structures with intrinsic sharing.~~. 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 ~~upgrade Hobor and Villard's theory of ramification to~~ develop a Localize rule that enables modular reasoning about such programs, and show how this rule can support existential quantifiers in postconditions and ~~to~~ 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 ~~six~~ 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:
. 2019. Certifying Graph-Manipulating C Programs via Localizations within Data Structures. In *Proceedings of Proceedings of the ACM on Programming Languages, Volume 3, Number OOPSLA (OOPSLA '19).* ACM, New York, NY, USA, 43 pages.

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.~~ *i.e.* structures with *intrinsic sharing*) have proved harder to verify. Indeed, such programs were formidable enough that ~~a number~~ 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 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 ~~general~~ toolkit for verifying graph-manipulating programs in a machine-checked context. Our techniques are *general* in that they handle a diverse range of ~~graph-manipulating~~ programs, and *modular* in that they ~~allow~~ encourage code reuse (*e.g.* facts about reachability) and

OOPSLA '19, October 20–25, 2019, Athens, Greece

~~encourage~~ separation of concerns (*e.g.* between abstract mathematical graphs and ~~their concrete representation~~ 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 ~~requiring~~ major reengineering. ~~Both~~ CompCert and VST are distributed ~~as optional packages in~~ via opam and the CoqIDE installer~~and through opam~~, so they have a sizable ~~user base~~ 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] ($\approx$ 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. ~~Finally~~Third, we develop a notion of *localization blocks* that ~~allows us to carry out~~ enables modular reasoning in our Hoare proofs~~in a modular fashion~~, 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. ~~The key is carefully choosing a *ramification frame* R that satisfies a pair of delicately-stated entailments[1] and the side condition on modified local program variables.~~ 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. ~~Said differently, Frame works well for tree-manipulating programs, while~~ Localize ~~handles the more subtle graph-manipulating programs~~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 ~~[Cormen et al. 2009]~~ 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 ~~introduce *localization blocks* as a notation for Localize in decorated programs~~ briefly review the seven programs we have already verified to give a sense of the breadth of algorithms our system can tackle.
- §3 We show that Localize and Frame are ~~equivalent, and~~ co-derivable. We illustrate a delicate technique to properly handle modified local variables. We show a mark-graph program

~~[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 16), 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]Readers less familiar with the separating implication $P \mathrel{-\!\!*} Q$, also known as *magic wand*, can refer to its semantics in Figure 8 (page 16), 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: $(P * Q \vdash R) \Leftrightarrow (P \vdash Q \mathrel{-\!\!*} R)$.

that explores a graph in a fold/unfold style, and discuss the utility of linked existentials in postconditions.
~~We also briefly discuss some additional examples to give a sense of the breadth of algorithms we can verify: marking a DAG, an array-based version of union-find, and pruning a graph into a spanning tree.~~

§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~~and show how our framework is modularized to facilitate code reuse~~.

§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 enjoys a "recursive" fold/unfold. We prove general theorems about spatial graphs ~~in a~~ 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). ~~While certifying this GC we~~ We identify two places where the semantics of C is too weak to define an OCaml-style GC. We also ~~found and fixed~~ find and fix a rather subtle overflow error in the original C code for the GC, thereby justifying the effort of developing ~~the~~ machine-checked ~~proof~~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 ~~available at [?]~~are available as an artifact online. A version of this paper featuring three additional appendices is available at [Wang et al. 2019].

2 ~~LOCALIZATIONS YIELD A TIDY UNION-FIND~~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 a well-known 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`~~ `find` function from the classic disjoint-set data structure [Cormen et al. 2009] in Figure 1. The ~~`find`~~ function returns the root (ultimate parent) of a ~~Node x.~~`Node x`. A node is a root ~~when its~~ whose parent pointer points to itself (line 11)~~; other~~. Other than such self-loops at roots, the structure is acyclic. For good amortised performance, ~~`find` also~~ `find` performs path compression (line 16). ~~At first, `find` appears rather trivial since it only has about 5 lines of code and a `Node` has only a single outgoing pointer. In actual fact, the rather~~ 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~~make the disjoint-set data structure very difficult to reason about~~. 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 ~~$\gamma(v) = (r, p)$~~ $\gamma(v) = (r, pa)$ to state that vertex v has label r and parent vertex ~~p~~ pa (r stores the "rank" of a node; it is ignored in ~~`find`~~`find`). We detail mathematical graphs in §4.

1 `struct Node { unsigned int rank;`

2 `struct Node * parent; }`

3 // $\{\mathsf{uf\_graph}(\gamma) \wedge \mathtt{x} \in V(\gamma)\}$

4 `struct Node* find(struct Node* x) {`

5 `struct Node *p;`

6 // $\left\{\begin{array}{l}\mathsf{uf\_graph}(\gamma) \wedge \mathtt{x} \in V(\gamma) \wedge \\ \exists r, pa.\ \gamma(\mathtt{x}) = (r, pa) \wedge pa \in V(\gamma)\end{array}\right\}$

7 // $\searrow$ $\left\{\begin{array}{l}\mathtt{x} \mapsto r, pa \wedge \mathtt{x} \in V(\gamma) \wedge \\ \gamma(\mathtt{x}) = (r, pa) \wedge pa \in V(\gamma)\end{array}\right\}$

8 $\lightning(2)$ `p = x -> parent;`

9 // $\swarrow$ $\left\{\begin{array}{l}\mathtt{x} \mapsto r, pa \wedge \mathtt{p} = pa \wedge \mathtt{x} \in V(\gamma) \wedge \\ \gamma(\mathtt{x}) = (r, pa) \wedge pa \in V(\gamma)\end{array}\right\}$

10 // $\left\{\begin{array}{l}\mathsf{uf\_graph}(\gamma) \wedge \mathtt{p} = pa \wedge \mathtt{x} \in V(\gamma) \wedge \\ \gamma(\mathtt{x}) = (r, pa) \wedge pa \in V(\gamma)\end{array}\right\}$

11 `if (p != x) {`

12 // $\left\{\begin{array}{l}\mathsf{uf\_graph}(\gamma) \wedge \mathtt{p} = pa \wedge pa \neq \mathtt{x} \wedge \\ \mathtt{x} \in V(\gamma) \wedge \gamma(\mathtt{x}) = (r, pa) \wedge pa \in V(\gamma)\end{array}\right\}$

13 `p = find(p);`

14 // $\left\{\begin{array}{l}\exists \gamma', rt.\ \mathsf{uf\_graph}(\gamma') \wedge \mathtt{p} = rt \wedge pa \neq \mathtt{x} \wedge \mathtt{x} \in V(\gamma) \wedge \\ findS(\gamma, pa, \gamma') \wedge uf\_root(\gamma', pa, rt) \wedge \gamma(\mathtt{x}) = (r, pa)\end{array}\right\}$

15 // $\searrow$ $\left\{\begin{array}{l}\mathtt{x} \mapsto r, pa \wedge \mathtt{p} = rt \wedge pa \neq \mathtt{x} \wedge findS(\gamma, pa, \gamma') \wedge \\ uf\_root(\gamma', pa, rt) \wedge \mathtt{x} \in V(\gamma) \wedge \gamma(\mathtt{x}) = (r, pa)\end{array}\right\}$

16 $\lightning(3)$ `x -> parent = p;`

17 // $\swarrow$ $\left\{\begin{array}{l}\mathtt{x} \mapsto r, rt \wedge \mathtt{p} = rt \wedge pa \neq \mathtt{x} \wedge findS(\gamma, pa, \gamma') \wedge \\ uf\_root(\gamma', pa, rt) \wedge \mathtt{x} \in V(\gamma) \wedge \gamma(\mathtt{x}) = (r, pa)\end{array}\right\}$

18 // $\left\{\begin{array}{l}\exists \gamma''.\ \mathsf{uf\_graph}(\gamma'') \wedge findS(\gamma, pa, \gamma'') \wedge \\ uf\_root(\gamma'', \mathtt{x}, rt) \wedge \mathtt{p} = rt\end{array}\right\}$

19 `} return p;`

20 `}` // $\left\{\begin{array}{l}\exists \gamma'', rt.\ \mathsf{uf\_graph}(\gamma'') \wedge findS(\gamma, \mathtt{x}, \gamma'') \wedge \\ uf\_root(\gamma'', \mathtt{x}, rt) \wedge \mathtt{ret} = rt\end{array}\right\}$

$$\mathsf{uf\_graph}(x, \gamma) \triangleq \mathop{\Large\ast}_{v \in V(\gamma)} v \mapsto \gamma(v)$$

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

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

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

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 ~~use~~ employ the usual pen-and-paper ~~trick~~ shorthand of writing *e.g.* ~~$v \mapsto r, p$ to mean~~ $v \mapsto r, pa$ to mean ~~$(v \mapsto r) * ((v + sizeof(\mathtt{unsigned\ int})) \mapsto p)$~~ $(v \mapsto r) * ((v + sizeof(\mathtt{unsigned\ int})) \mapsto pa)$ 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~~shortly~~.

The precondition, stated on line 3, says that we have a disjoint-set forest representing the abstract graph γ, and that ~~x~~ 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 pure~~ 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: ~~$x \overset{\gamma}{\leadsto}{}^{\star} y$~~ $x \overset{\gamma}{\leadsto}{}^{\star} y$ 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. ~~To aid human~~ For readability, we ~~use the color~~ mark line-by-line changes in the invariants in red~~to indicate the changes in the invariants line-by-line. Each individual~~. 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—that is,~~ 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 ~~$(x \mapsto \_) * (p \mapsto \_) * \top$~~$(\mathsf{x} \mapsto \_) * (\mathsf{p} \mapsto \_) * \top$, but unfortunately this does not follow from line 9 since, when ~~x~~ x is a root, self-loop gives us ~~$x = p$ and $x \mapsto \_ * x \mapsto \_ \vdash \bot$~~$\mathsf{x} = \mathsf{p}$ and $\mathsf{x} \mapsto \_ * \mathsf{x} \mapsto \_ \vdash \bot$. Accordingly, we must prove a simple lemma that states that when ~~$\mathsf{uf\_graph}(\gamma) \wedge x \in V(\gamma) \wedge p \in V(\gamma)$~~$\mathsf{uf\_graph}($ 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~~ $\searrow$, $\lightning(i)$, 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. ~~Intuitively~~To give intuition, a localization block allows us to zoom in from a larger "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. ~~The $\searrow$ and $\swarrow$ symbols formally indicate an application of the Localize rule (equation 1 from page 2).~~

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 $\forall P.\ (P \dashVdash \exists x : unit.P)$ for any x not free in P.

~~What is not specified explicitly is the~~ We must now pick a ramification frame R ~~, which must satisfy~~ that satisfies the entailments $G_1 \vdash L_1 * R$ and (again eliding the quantifier) $R \vdash L_2 \mathrel{-\!\!*} G_2$. ~~Here things are a little~~ 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 ~~reduces the problem to three remaining~~ leaves us with three checks.

The first check is the *ramification entailment*~~:~~, $G_1 \vdash L_1 * (L_2 \mathrel{-\!\!*} G_2)$~~. Informally, this~~, which asks whether replacing L_1 with L_2 inside G_1 yields G_2. This ~~ramification entailment~~ is a nontrivial proof obligation both because we must prove that L_1 is located inside G_1, and ~~because we must prove~~ that "replacing" L_1 with L_2 yields G_2. ~~When we want to~~ Henceforth, we refer to the ramification entailment of a localization block ~~in subsequent text we use~~ by using the symbol $\lightning(i)$ to connect it to ~~relevant equation numbers~~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

[2] ~~To summarize~~Executive summary: it is a mess. ~~Specifically,~~ Full story: whenever (1) ~~x~~ x and ~~p~~ 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~~ x and ~~p~~ 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~~ x and ~~p~~ p are not in the same memory block and both have offsets between 0 and the size of their respective memory blocks *minus one*.

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~~ `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 ~~does contain~~ 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. ↯(3) expresses an update to a single node of our graph:

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

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~~, as will be explained in §4.1.

~~In our proof of , the bulk of the~~ The example-specific ~~effort (as opposed to generic lemmas we reuse in other examples)~~ challenge in proving `find` is showing that this ~~mathematical~~ update can be done properly, *i.e.* from

$$\mathsf{x} \in V(\gamma) \land \gamma(\mathsf{x}) = (r, pa) \quad \text{and} \quad \mathsf{x} \neq pa \land \mathit{findS}(\gamma, pa, \gamma') \land \mathit{uf\_root}(\gamma', pa, rt)$$

~~we can 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 lemma captures the essence of both finding the root and doing path compression~~
$\mathsf{x} \in V(\gamma) \land \gamma(\mathsf{x}) = (r, pa)$ 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, ~~you can path~~ 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 graph copying algorithm, which is tricky because we must initially reason about a "copy" that is under construction, but eventually show isomorphism with the well-formed original. Further, 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) ~~first~~ 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. ~~Finally (§2.3), we briefly discuss some additional examples we have handled. Our flagship example, the garbage collector for the CertiCoq project, will be covered in §6.~~

3.1 Soundness of Localize

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

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

~~Proving Localize from Frame, and conversely Frame from Localize~~

In Figure 2 we put ~~the~~ 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

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

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

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

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

restriction on modified program variables is satisfied: in the first proof, Localize's side condition that $\mathrm{FreeVar}(R) \cap \mathrm{ModVar}(c) = \varnothing$ is exactly what Frame needs; in the second, Frame's side condition that $\mathrm{FreeVar}(F) \cap \mathrm{ModVar}(c) = \varnothing$ 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.

~~*Notes on notation.* Although we do not do so in Figure 1, localization~~ *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 ~~can~~ write $\{G_1\} \searrow \{L_1\}$ and $\{G_2\} \swarrow \{L_2\}$. ~~We also note that since~~ Since Localize can derive Frame, our notation ~~for localization blocks~~ also clarifies pen-and-paper uses of Frame, especially in multi-line contexts with nontrivial frame F.

3.2 Smoothly ~~handling modified program variables~~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, ~~then~~ the program variable x appears in all four positions in the ramification entailment

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

For the sake of simplicity, assume that in the above snippet only ~~x~~ x is modified and that ~~x~~ 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 ~~seems to be~~ is a bit too strong since it prevents us from mentioning

[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.

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 $\hat{L_2}(x_f) \triangleq (x_f = 6 \wedge C)$ and $\hat{G_2}(x_f) \triangleq (x_f = 6 \wedge D)$, *i.e.* replace the troublesome program variable ~~x~~ 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 isolatedand shift the above decorated program into the form~~ x isolated. The new decorated program is below.

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

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 ~~$\hat{\hat{L_2}}(x_f) \triangleq (x_f = \mathtt{x}) \wedge \hat{L_2}(x_f)$~~ $\hat{\hat{L_2}}(x_f) \triangleq (x_f = \mathtt{x}) \wedge \hat{L_2}(x_f)$, *i.e.* ~~$(x_f = \mathtt{x}) \wedge (x_f = 6 \wedge C)$ and~~ $(x_f = \mathtt{x}) \wedge (x_f = 6 \wedge C)$. $\hat{\hat{G_2}}(x_f)$ is similar. Now apply Localize with $R \triangleq \forall x_f.\ \hat{L_2}(x_f) \mathrel{-\!\!*} \hat{G_2}(x_f)$, *i.e.* $\forall x_f.\ (x_f = 6 \wedge C) \mathrel{-\!\!*} (x_f = 6 \wedge D)$. 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.*

$$\forall x_f.\ \big(\hat{L_2}(x_f) \mathrel{-\!\!*} \hat{G_2}(x_f)\big) \vdash \forall x_f.\ \Big(\big((\underline{\mathtt{x}}\mathtt{x} = x_f) \wedge \hat{L_2}(x_f)\big) \mathrel{-\!\!*} \big((\underline{\mathtt{x}}\mathtt{x} = x_f) \wedge \hat{G_2}(x_f)\big)\Big)$$

This ~~turns out to be just~~ is a long-winded tautology, and ~~can be done automatically~~ is automatic in a tool.

~~The first of Localize's entailments~~ The first entailment is $G_1 \vdash L_1 \mathrel{-\!\!*} R$, *i.e.*

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

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

With ~~carefully engineering, all of the~~ careful engineering, this modified-variable dance ~~above~~ can be done fully automatically, ~~and~~ in a way that is ~~completely hidden to~~ hidden from end-users. The only remaining proof goal is the spatial part, which captures the key action of the localization block. ~~To solve these in~~ In practice, we ~~end up applying~~ 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.

~~In a mechanized context, working with existing toolsets, these kinds of~~ These solutions are not viable when using existing toolsets in a mechanized context. Either we must reinvent a **very**~~, **very**~~ large wheel—combined, VST and CompCert ~~contain~~ 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, ~~it is hardly unique in these respects:~~ most other mechanized verification systems [Beckert et al. 2007; Bengtson et al. 2012; Chin et al. 2010; Distefano and Parkinson 2008]

```
1   struct Node { int _Alignas(16) m;
2            struct Node * _Alignas(8) l, * r };
3   void mark(struct Node * x) {
4   // {m_graph(x, γ)}
5     struct Node * l, * r; int root_mark;
6     if (x == 0) return;
7   // {m_graph(x, γ) ∧ ∃m, l, r.γ(x) = (m, l, r)}
8   // {m_graph(x, γ) ∧ γ(x) = (m, l, r)}
9   //↘ {x ↦ m, −, l, r}
10    root_mark = x -> m;
11  //↙ {x ↦ m, −, l, r ∧ m = root_mark}
12  // {m_graph(x, γ) ∧ γ(x) = (m, l, r) ∧ m = root_mark}
13    if (root_mark == 1) return;
14  // {m_graph(x, γ) ∧ γ(x) = (0, l, r)}
15  //↘ {x ↦ 0, −, l, r ∧ γ(x) = (0, l, r)}
16  ↯(8)  l = x -> l; r = x -> r; x -> m = 1;
17  //↙ {x ↦ 1, −, l, r ∧ γ(x) = (0, l, r) ∧
           ∃γ'.mark1(γ, x, γ')}
18  // {∃γ'.m_graph(x, γ') ∧ γ(x) = (0, l, r) ∧ mark1(γ, x, γ')}
19  // {m_graph(x, γ') ∧ γ(x) = (0, l, r) ∧ mark1(γ, x, γ')}
20  //↘ {m_graph(l, γ')}
21  ↯(7)  mark(l);
22  //↙ {∃γ''.m_graph(l, γ'') ∧ mark(γ', l, γ'')}
23  // {∃γ''.m_graph(x, γ'') ∧ γ(x) = (0, l, r) ∧
        mark1(γ, x, γ') ∧ mark(γ', l, γ'')}
24  // {m_graph(x, γ'') ∧ γ(x) = (0, l, r) ∧
        mark1(γ, x, γ') ∧ mark(γ', l, γ'')}
25  //↘ {m_graph(r, γ'')}
26  ↯(7)  mark(r);
27  //↙ {∃γ'''.m_graph(r, γ''') ∧ mark(γ'', r, γ''')}
28  // {∃γ'''.m_graph(x, γ''') ∧ γ(x) = (0, l, r) ∧
        mark1(γ, x, γ') ∧ mark(γ', l, γ'') ∧ mark(γ'', r, γ''')}
29  } // {∃γ'''.m_graph(x, γ''') ∧ mark(γ, x, γ''')}
```

$$
\begin{aligned}
\mathrm{m\_graph}(x,\gamma) \;\Leftrightarrow\; & (x = 0 \wedge \mathrm{emp}) \vee \\
& \exists m, l, r \,.\, \gamma(x) = (m, l, r) \wedge x \bmod 16 = 0 \wedge \\
& \quad x \mapsto m, -, l, r \uplus \mathrm{m\_graph}(l,\gamma) \uplus \mathrm{m\_graph}(r,\gamma)
\end{aligned}
\qquad (4)
$$

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

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

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

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

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

do not support these solutions either ~~[Beckert et al. 2007; Bengtson et al. 2012; Chin et al. 2010; Distefano and~~ ~~. By respecting the design decisionstaken by most existing tools~~. 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~~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. ~~Lastly~~Fourth, the specification we certify (lines 3 and 29) is *local* rather than *global*:

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

The specification is again stated with mathematical γ, although ~~in this case $\gamma(x)$~~ $\gamma(\mathrm{x})$ 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 ~~v~~x. By "local", we mean that the predicate $\mathrm{m\_graph}(\mathrm{x},\gamma)$ says that the heap represents *only the nodes* *in γ that are reachable from* x. Rather than passing the entire graph around as `find` does, `mark` ~~will use~~ uses the fold/unfold relationship given in~~equation~~ (4) ~~, located just under the code in Figure 3,~~ to "unfold" the graph as if it were an inductive predicate.

This fold/unfold relationship deserves attention. ~~First, uses the~~ Note the use of the "overlapping conjunction" $\uplus$ of separation logic; informally $P \uplus Q$ 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 ~~presence of the~~ unspecified sharing indicated by the $\uplus$ 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) ~~illustrates~~ 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 Floyd proofs~~In our proofs, the alignment restriction and blank second field are ~~nicely hidden "~~neatly hidden behind the scenes~~".)~~.

Just as with `find`, the postcondition of `mark` is specified *relationally*, *i.e.* ~~$\{\exists \gamma'.\ \mathsf{graph}(x, \gamma') \wedge \mathit{mark}(\gamma, x, \gamma')\}$~~ $\{\exists \gamma'.\ \mathsf{m\_graph}(x, \gamma') \wedge \mathit{mark}(\gamma, x, \gamma')\}$ instead of *functionally*, *i.e.* ~~$\{\mathsf{graph}(x, \mathit{mark}(\gamma, x))\}$~~ $\{\mathsf{m\_graph}(x, \mathit{mark}(\gamma, x$ 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 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~~ terminate—a nontrivial proof obligation over cyclic structures like ~~graphs, but~~ 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.

```
// {A} ↘ {B}
      ...; x = malloc(sizeof(int));
      if (x == 0) then y = 0 else y = 1; ...;
// ↙ {((x ↦ − ∧ y = 1) ∨ (x = 0 ∧ y = 0)) ∗ C}
// {(y = 1 ∧ D1) ∨ (y = 0 ∧ D2)}

// ↙ {graph(l, γ'') ∧ mark(γ', l, γ'')}
```

3.4 ~~Additional verified examples~~

[4] ~~Recall that the~~ The standard semantics of the separation logic connectives used in this paper are in Figure 8 on page 16.

~~In addition to the proof of `find` shown above, we do a proof of `union` in the same style. We also do a *second* version of union-find, this time using arrays rather than `malloc`-allocated nodes. The mathematical definitions, *e.g.* for *findS*, are entirely shared, demonstrating that we have separated the concerns of abstract algorithmic reasoning from the nitty-gritty details of heap representation.~~

~~We also have done a version of `mark` for DAGs (acyclic). In general, using DAGS are both easier and harder than cyclic graphs. On the one hand, we get genuine separation between the root and its children; on the other hand, we need to maintain acyclicity if we modify the link structure. As a more aggressive example of modifying the link structure of a graph, we have verified , which prunes a graph into its spanning tree; for space reasons we put this example in Appendix A. Our flagship example, the CertiCoq garbage collector, will be discussed in §6.~~

4 A REUSABLE LIBRARY OF FORMALIZED GRAPH THEORY

~~To verify~~ When verifying the functional correctness of graph algorithms it is natural to want to use graph theory to describe program behavior. ~~As discussed in §8,~~ Over the past 25 years~~of research into mechanized graph theory can be succinctly summarized as "it is a little tricky".~~ , 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 ~~is actually powerful enough, and expressive enough, to mechanically verify realistic algorithms written in C~~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. ~~We~~ In this section we present our mathematical graph framework with an emphasis on this modularity.

4.1 ~~Definitions of graphs~~

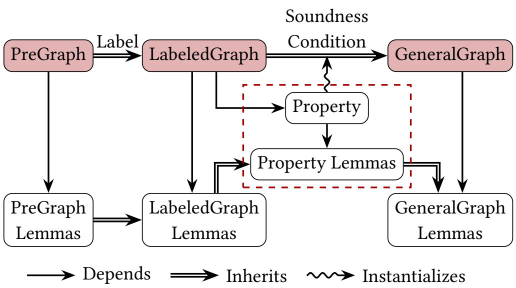

Fig. 4. Structure of the Mathematical Graph Library

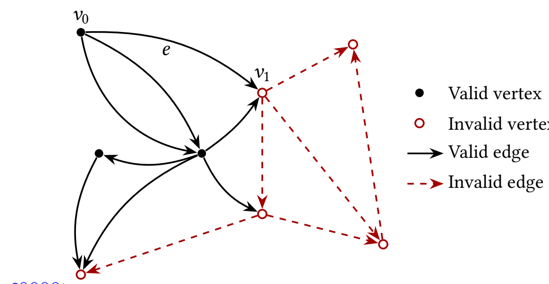

Fig. 5. PreGraph with valid/invalid vertices/edges.

$$\text{path} \triangleq (v_0, [e_0, e_1, \ldots, e_k])$$

$$\texttt{s\_evalid}(\gamma, e) \triangleq \mathsf{E}_\gamma(e) \wedge \mathsf{V}_\gamma(\texttt{src}_\gamma(e)) \wedge \mathsf{V}_\gamma(\texttt{dst}_\gamma(e))$$

$$\texttt{valid\_path}\big(\gamma, (v, [])\big) \triangleq \mathsf{V}_\gamma(v)$$

$$\begin{aligned}\texttt{valid\_path}\big(\gamma, (v, [e_1, e_2, \ldots, e_n])\big) \triangleq\ & v = \texttt{src}_\gamma(e_1) \wedge \texttt{s\_evalid}(\gamma, e_1) \wedge \\ & \texttt{dst}_\gamma(e_1) = \texttt{src}_\gamma(e_2) \wedge \\ & \texttt{s\_evalid}(\gamma, e_2) \wedge \ldots \wedge \texttt{dst}_\gamma(e_{n-1}) = \texttt{src}_\gamma(e_n)\end{aligned}$$

$$\texttt{end}\big(\gamma, (v, [])\big) \triangleq v$$

$$\texttt{end}\big(\gamma, (v, [e_1, e_2, \ldots, e_n])\big) \triangleq \texttt{dst}_\gamma(e_n)$$

$$\gamma \vDash s \overset{p}{\rightsquigarrow} t \triangleq \texttt{valid\_path}(\gamma, p) \wedge \texttt{fst}(p) = s \wedge \texttt{end}(\gamma, p) = t$$

$$\begin{aligned}\gamma_1 \cong \gamma_2 \triangleq\ & \forall e.\ \mathsf{E}_{\gamma_1}(e) \Leftrightarrow \mathsf{E}_{\gamma_2}(e) \wedge \forall v.\ \mathsf{V}_{\gamma_1}(v) \Leftrightarrow \mathsf{V}_{\gamma_1}(v) \wedge \\ & \forall e.\ \mathsf{E}_{\gamma_1}(e) \Rightarrow \texttt{src}_{\gamma_1}(e) = \texttt{src}_{\gamma_2}(e) \wedge \texttt{dst}_{\gamma_1}(e) = \texttt{dst}_{\gamma_2}(e)\end{aligned}$$

$$\begin{aligned}\gamma \setminus S \triangleq\ & \mathcal{V}_{\gamma'} = \mathcal{V}_\gamma \wedge \mathcal{E}_{\gamma'} = \mathcal{E}_\gamma \wedge \\ & \texttt{src}_{\gamma'} = \texttt{src}_\gamma \wedge \texttt{dst}_{\gamma'} = \texttt{dst}_\gamma \wedge \\ & \gamma'_{\mathsf{V}} = \Big(\lambda x.\ \gamma_{\mathsf{V}}(x) \wedge \neg S_{\mathsf{V}}(x)\Big) \wedge \\ & \gamma'_{\mathsf{E}} = \Big(\lambda x.\ \gamma_{\mathsf{E}}(x) \wedge \neg S_{\mathsf{E}}(x)\Big)\end{aligned}$$

$$\begin{aligned}\text{MathGraph}(\gamma) \triangleq \Big\{ & \texttt{null} : V \\ \texttt{valid\_graph} :\ & \forall e\,.\,\texttt{evalid}(\gamma, e) \Rightarrow \texttt{vvalid}\big(\gamma, \texttt{src}(\gamma, e)\big) \wedge \\ & \big(e = \texttt{null} \vee \texttt{vvalid}(\gamma, e)\big) \\ \texttt{valid\_not\_null} :\ & \forall v\,.\,\texttt{vvalid}(\gamma, v) \Rightarrow v \neq \texttt{null} \Big\}\end{aligned}$$

$$\begin{aligned}\text{LstGraph}(\gamma) \triangleq \Big\{ & \texttt{out} : V \rightarrow E \\ \texttt{only\_one\_edge} :\ & \forall v,\ e\,.\,\texttt{vvalid}(\gamma, v) \Rightarrow \\ & \Big(\texttt{src}(\gamma, e) = v \wedge \texttt{evalid}(\gamma, e)\Big) \Leftrightarrow e = \texttt{out}(v) \\ \texttt{acyclic\_path} :\ & \forall v,\ p\,.\,\gamma \vDash v \overset{p}{\rightsquigarrow} v \Rightarrow p = (v, []) \Big\}\end{aligned}$$

$$\begin{aligned}\text{FiniteGraph}(\gamma) \triangleq \Big\{ & \texttt{finite\_v} : \exists S_v,\ M_v \text{ s.t. } |S_v| \leq M_v \wedge \forall v\,.\,\texttt{vvalid}(\gamma, v) \Rightarrow v \in S_v \\ & \texttt{finite\_e} : \exists S_e,\ M_e \text{ s.t. } |S_e| \leq M_e \wedge \forall e\,.\,\texttt{evalid}(\gamma, e) \Rightarrow e \in S_e \Big\}\end{aligned}$$

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

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

Fig. 6. Some PreGraph definitions

~~Some Graph definitions~~

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*. ~~We choose to formalize graph theory directly in Coq instead~~ Instead of formalizing set theory in Coq and then building graph theory atop it~~to~~, we formalize graph theory directly in Coq, as this lets us take advantage of Coq's built-in support for type-related constructions. To ~~balance~~ reconcile the dichotomy between ~~the generality and the speciality of the library , we divide the concept of graph into three structures,~~ a very general library and highly specialized examples, we develop our graphs gradually over three linked concepts: PreGraph, LabeledGraph and GeneralGraph~~, arranged in a hierarchy~~. Figure 4 shows the architecture of the library.

PreGraph. A PreGraph is a hextuple $(\mathcal{V}, \mathcal{E}, \mathsf{V}, \mathsf{E}, \mathtt{src}, \mathtt{dst})$. 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` map each edge to its source and destination.

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~~; *e.g.*,~~, thus leaving the edge e ~~is~~ 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

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

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

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

Fig. 7. Some GeneralGraph definitions

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. ~~The need for such labels is that many~~ Many classic graph problems, from union-find (node ranks) to Dijkstra (edge weights)~~require them~~, require such labels. The need for a label on the graph as a whole is ~~a little more subtle~~not as obvious; in §6 we use one in the garbage collector to keep track of ~~e.g.~~ the number of generations and their boundaries. Since every LabeledGraph is built on a PreGraph, it inherits all ~~PreGraph~~ of the PreGraph's lemmas via Coq's *type coercion* mechanism ~~, while enabling~~ 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` restricted~~ `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 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 ~~a~~ its internal LabeledGraph or a PreGraph, thereby inheriting their lemmas. This combination of specificity and generality ~~is what~~ 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 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

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

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

- 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*: ~~all valid edges must have~~ every valid edge has a valid source vertex~~; the destination vertex must either be valid or be~~ , and its destination vertex is either valid or is a special invalid ~~node~~ vertex called null
- FiniteGraph*: the sets of valid vertices and edges are both finite
- ~~LstGraph*: the graph is list-like: each vertex has only one outgoing edge; no nontrivial loops~~
- ~~: there are exactly two outgoing edges per node~~

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`~~ `find` in Figure 1. In our verification of ~~`mark`~~ `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

$$
\begin{array}{lcl}
\sigma \models P * Q & \triangleq & \exists \sigma_1, \sigma_2.\, (\sigma_1 \oplus \sigma_2 = \sigma) \wedge (\sigma_1 \models P) \wedge (\sigma_2 \models 2) \\
\sigma \models P \mathrel{-\!\!*} Q & \triangleq & \forall \sigma_1, \sigma_2.\, (\sigma_1 \oplus \sigma = \sigma_2) \wedge (\sigma_1 \models P) \Rightarrow (\sigma_2 \models Q) \\
\sigma \models P \mathrel{-\!\!\circledast} Q & \triangleq & \exists \sigma_1, \sigma_2.\, (\sigma_1 \oplus \sigma = \sigma_2) \wedge (\sigma_1 \models P) \wedge (\sigma_2 \models Q) \\
\sigma \models P \uplus Q & \triangleq & \exists \sigma_1, \sigma_2, \sigma_3.\, (\sigma_1 \oplus \sigma_2 \oplus \sigma_3 = \sigma) \wedge (\sigma_1 \oplus \sigma_2 \models P) \wedge (\sigma_2 \oplus \sigma_3 \models Q)
\end{array}
$$

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

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 ~~the~~ (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~~as we shall see in §5.2 and §5.3~~: 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 ~~for the others~~ thereafter is to engineer a framework that is generic ~~enough~~ and modular enough to be useful in ~~practice in~~ a variety of settings~~; we cover it in §5.4~~ (§5.4).

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'H 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.

$$\mathsf{tree}(x,\tau) \triangleq \Big(x = 0 \land \mathsf{isatom}(\tau) \land \mathsf{emp}\Big) \lor \\ \Big(\exists\, l, r, \tau_1, \tau_2.\ \tau = \langle \tau_1, \tau_2 \rangle \land (x \mapsto l, r) * \mathsf{tree}(l, \tau_1) * \mathsf{tree}(r, \tau_2)\Big) \tag{5}$$

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.

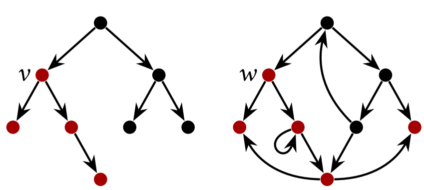

Fig. 9. A Binary Tree and a Binary Graph

address	value
102	0
101	100
100	42

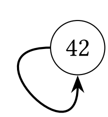

Fig. 10. A One-Cell Graph and its Heap Layout

5.2 Recursive ~~definitions yield poor~~ Definitions Yield Poor graph ~~predicates~~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 Knaster-Tarski fixpoint μ_{T} [Tarski 1955]. Formally, one does not define P directly, but rather defines a functional $F_P \triangleq \lambda P.\ \ldots P \ldots$ and then defines P itself as $P \triangleq \mu_{\mathrm{T}}\, F_P$. Assuming, as one typically does without comment, that F_P is *covariant*, i.e. $\cancel{(P \vdash Q) \Rightarrow (F\,P \vdash F\,Q)}(P \vdash Q) \Rightarrow (F\,P \vdash F\,Q)$, one then enjoys the fixpoint equation $P \Leftrightarrow\ \ldots P \ldots$, formally justifying the typically-written pseudodefinition ("$\triangleq$").

Suppose we define a graph predicate graph_T ~~this way, *e.g.*~~ 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:

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

Although we can apply Knaster-Tarski (because the functional needed to define graph_T is covariant), the result is hard to use. Consider the ~~following~~ memory m for a toy machine~~:~~

address	value
102	0
101	100
100	42

~~Clearly~~ , where $m \models 100 \mapsto 42, 100, 0$. ~~But it seems also clear~~It seems clear, however, that this memory also represents a one-cell cyclic graph as illustrated in ~~the accompanying diagram~~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 ~~explained in Appendix C~~illustrated in Figure 11, this is rather difficult to do since applying the natural proof techniques actually strengthens the goal. ~~In fact we~~ 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.

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

x † 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

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

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

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:

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

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

5.3 Defining a ~~good~~ Good graph ~~predicate~~Predicate

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

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

We are now ready to define a good graph predicate: ~~$\mathrm{graph}(x,\gamma) \triangleq \mathop{\Large\ast}_{v \in \mathit{reach}(\gamma,x)} v \mapsto \gamma(v)$~~ $\mathrm{graph}(x,\gamma) \triangleq \mathop{\Large\ast}_{v \in \mathit{reachab}}$

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):

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

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 (~~16~~ *e.g* this size is 16 in Figure 3).

Some of our VST proofs, *e.g.* for ~~`find`~~`find`, do not use fold/unfold, instead preferring to use the lemmas in §5.4 directly. Others, *e.g.* ~~`mark`, `do` and~~ `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

~~VST employs a somewhat unusual~~ Many tools (*e.g.* Charge! [Bengtson et al. 2012], Smallfoot [Berdine et al. 20 , 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 ~~to represent spatial predicates, and uses it to~~ in order to support an unusually rich program

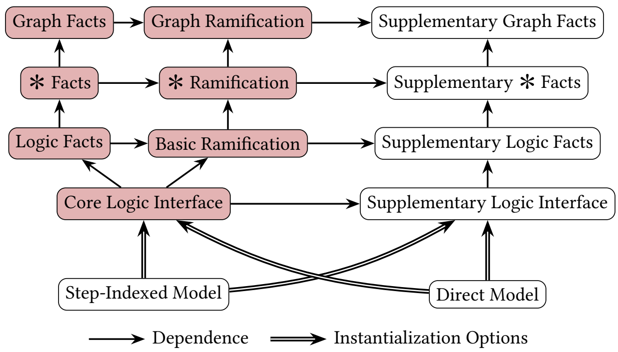

Fig. 12. Infrastructure of ramification library

logic~~. [Appel et al. 2014]However, the spatial representation of our graph library does~~ [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 ~~its bellsand whistles. To isolate our development from these unnecessary complications, we use~~ 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~~, where Supplementary Logic is built upon the much simpler and more universal Direct Model of heap representation. We present the architecture of our spatial development in Figure 12, where it should be straighforward to see that both models can instantiate both interfaces.~~

~~Generally speaking, our VST proofs only need the Core properties to prove our examples.~~. Each interface defines some operators of separation logic ~~and provides some axioms about~~ along with relevant axioms to show how they work. For example, the definitions of $*$ and $-\!\!*$ are in Core Logic, along with ~~the axiom~~ key axioms and rules such as $(P \vdash Q -\!\!* R) \Leftrightarrow (P * Q \vdash R)$. On the other hand, ~~the~~ lesser-used operators such as $\uplus$ and $-\!\!\circledast$ ~~operators~~ are in Supplementary Logic, along with rules ~~like~~ such as $P \vdash P \uplus P$. 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, $\circledast$, and graph. For instance, in the $\circledast$ Facts box we prove:

$$\frac{A \cap B = \emptyset}{\underset{x \in A}{\circledast} P(x) * \underset{x \in B}{\circledast} P(x) \Leftrightarrow \underset{x \in A \cup B}{\circledast} P(x)}$$

$$\frac{A \cap B = \emptyset}{\underset{x \in A}{\circledast} P(x) * \underset{x \in B}{\circledast} P(x) \Leftrightarrow \underset{x \in A \cup B}{\circledast} P(x)}$$

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 ~~:~~

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

~~We use this lemma~~ the one below, which we use to break large ramification entailments into ~~more manageable piecesin a compositional way.~~ compositionally manageable pieces.

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

The $\circledast$ Ramification box in the middle layer contains lemmas like ~~:~~ the one below:

$$\frac{A \cap B = \varnothing \qquad A' \cap B = \varnothing}{\underset{x \in A \cup B}{\circledast} P(x) \vdash \underset{x \in A}{\circledast} P(x) * \left(\underset{x \in A'}{\circledast} P(x) \mathrel{-\!\!*} \underset{x \in A' \cup B}{\circledast} P(x) \right)}$$

~~The~~ 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 ~~in the top layeris focused lemmas such as the following "update one node" lemma , which was~~ (top layer), is used on lines 21 and 26 of Figure 3 to reason about smoothly replacing subgraphs.

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

Similarily, the same box contains the key lemma used on line 16 in Figure 3 ~~:~~

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

to update one node:

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

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 $\circledast$ 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. ~~Taking the marking algorithm we introduced in §2 as an example, we prove the following theorem from the library:~~

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

The Supplementary tower contains properties not used by most of the VST examples. This includes the fold/unfold relationship ~~from~~ 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 ~~uses the Direct Model.~~ needs separation logic facts about $\uplus$, $-\circledast$, *etc.*

6 CERTIFYING A GARBAGE COLLECTOR FOR CERTICOQ

6.1 ~~Background~~

The CertiCoq compiler [Anand et al. 2017] translates Gallina code to Clight, which CompCert [Leroy 2006] then compiles to assembly~~ [Leroy 2006]. CertiCoq's ~~. 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)~~is ~~, also written in Clight~~and supports Gallina's assumption of infinite memory~~. CertiCoq aims to be end-to-end certified, so the GC must ~~be too.~~ 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 ~~collector~~GC, written in the spirit of the OCaml GC, is ~~relatively ~~realistic and sophisticated, though by no means industrial-strength. Because CertiCoq ~~borrows ~~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~~, etc. The CertiCoq GC's task is a little ~~. 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 ~~, which ~~called `args`, and the GC scans this array to calculate the root set. ~~When called, the ~~The GC collects the first generation into the second using Cheney's algorithm [Cheney 1970]. This ~~collection may ~~may recursively trigger the collection of the second generation into the third, ~~etc., and the GC completes this potential cascade before returning ~~*etc.*, following which the GC returns control to the mutator. ~~A fuller explanation of ~~We provide further details about the GC's operation ~~is in Appendix ??~~in our extended online paper.

The mutator's ~~array of local variables is critical in ~~`args` array is critical to the GC's formal specification. The ~~GC must ensure that all ~~heap may change dramatically, but all heap memory objects that the mutator ~~can ~~could reach by recursively following the fields of `args` *before* the collection ~~can still be reached, via the same ~~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 ~~C ~~Clight to Mathematical Graphs

In the code, ~~the metainformation of the 12 generations is stored as an 12-element array . Each generation , as a memory segment, is represented as a , which contains three pointers : marking the start addressof a generation, representing the next available addressin a generation, and marking the ~~a generation is a contiguous memory segment. Three pointers capture key metainformation about a generation: `start` marks the first address, `limit`, the last address~~of a generation~~, 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 ~~garbage collector is a chunk ~~GC is a contiguous piece of memory called a ~~block. Blocks can be of different sizes; the size of a particular block~~*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~~(using 22 bits of the word stored at offset −*sizeof*(void*)), and the remainder~~

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

~~of the block is a continuous array of fields~~. Each field is either an unboxed integer data value ~~,~~ or a pointer~~, which may be either within the GC or to external structures~~. To disambiguate the two, we follow OCaml's practice of requiring that all ~~pointers are even-aligned and that all integers to be~~ integers are odd (essentially, ~~to be~~ 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.

~~From the perspective of the algorithm~~Moving towards a mathematical abstraction, the 12 generations can be seen as a graph γ. ~~Each block can be can be seen as a vertex~~ Blocks are vertices, and pointers to other blocks ~~indicate edges between vertices. More formally, we decide to encode each vertex of this graph as a pair of natural numbers~~ ~~(v_g, v_i)~~ ~~which means the vertex is the~~ ~~v_ith~~ are edges to other vertices. Vertices are tuples of the form (g, i), meaning the vertex represents the the i^{th} block in the ~~v_gth generation. We encode each edge as a pair of vertex and index~~ g^{th} generation. An edge is a tuple (v, i)~~which means this edge is from vertex~~ ~~v~~ ~~and~~, meaning that the associated pointer is ~~in the~~ ~~ith~~ the i^{th} field of the ~~corresponding block . The source function always satisfies~~ ~~$src(\gamma,(v,i)) = v$.~~ block corresponding to vertex v. Each vertex is labeled with ~~the integer dataitems and the indices of pointers in the fields~~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 ~~of~~ on the entire graph γ which has the start/limit addresses and number of vertices of each generation. ~~We can reconstruct the 12 generations in memory from~~ ~~γ~~ ~~under this setting without redundancy. For example, to determine the pointer of a generation, we can sum the sizes of each vertex in that generation using its label, and then add the address.~~

6.3 Forward

~~The function~~

The function `forward` is the GC's workhorse. When correctly given the spaces ~~and~~ `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 ~~of and~~ `struct thread_inf` and `file_info`—*finf* and *tinf*~~—together~~ —represent the graph's metainformation, and together allow us to extract the ~~mutator's array, and The proposition encapsulates various checksabout~~ `args` array. The proposition *compat* encapsulates a series of checks, *e.g.* ~~legal bounds that avoid some overflow issues. For concision,~~ 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 nth edge of vertex v. For readability, we denote the facts known to us in *e.g.* line 1 ~~are represented~~ by ϕ_1, and then use ϕ_1 ~~can feature~~ as a fact ~~*e.g.*~~ ~~in line 8~~in later annotations.

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

[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  // { ∀γ, finf, tinf, from, to, v, n.
        gc_graph(γ) ∗ finf(finf) ∗ tinf(tinf) ∧
        compat(γ, finf, tinf, from, to) ∧
        s = start(γ, from) ∧ l = s + gensz(γ, from) ∧
        n = nxtaddr(tinf, to) ∧ p = vaddr(γ, v) + n } ≜ ϕ1
2  void forward (value *s, *l, **n, *p) {
3    value *v; value va = *p;
4    if(Is_block(va)) {   //is ptr
5    v = (value *)((void *)va);
6    if(Is_from(s, l, v)) {   //in from
7  // { ϕ1 ∧ ∃e, v′. lab(γ, v)[n] = e ∧
        dst(γ, e) = v′ ∧ v = vaddr(γ, v′) } ≜ ϕ7
8  //↘ { ∃flds′, hdr′. flds′ = lab(γ, v′) ∧
        v′ ↦ flds′ ∧ hdr′ = flds′[−1] } ≜ ϕ8
9      header_t hd = Hd_val(v);
10 //↙ { ϕ8 ∧ hd = val(hdr′) }
11 // { ϕ7 ∧ hd = val(hdr′) }
12     if(hd == 0) {   //already forwarded
13 // { ϕ7 ∧ hd = 0 } ≜ ϕ13
14 //↘ { ∃flds, flds′. v ↦ flds ∧ v′ ↦ flds′ ∧
        flds = lab(γ, v) ∧ flds′ = lab(γ, v′) ∧
        flds′[0] = vaddr(γ, copy(γ, v′)) ∧
        p = &flds[n] } ≜ ϕ14
15     *p = Field(v,0);
16 //↙ { ϕ14 ∧ flds[n] := flds′[0] }
17 // { ϕ13 ∧ ∃γ′. gc_graph(γ′) ∧
        γ′ = upd_edge(γ, e, copy(γ, v′)) ∧
        fwd_postcondition(γ, γ′, tinf, finf, from, to, v, n) }
18     } else {   //not yet forwarded
19 // { ϕ7 ∧ hd ≠ 0 } ≜ ϕ19
20       int i; int sz; value *new;
21       sz = size(hd); new = *n+1;
22       *n = new+sz;
23 // { ϕ19 ∧ sz = blocksize(hd) ∧
        new = start(γ, to) + used(γ, to) + 1 ∧
        n = new + sz } ≜ ϕ23
24       Hd_val(new) = hd;
25       for(i = 0; i < sz; i++)
26       Field(new, i) = Field(v, i);
27 // { ϕ23 ∧ ∃γ′, v″, tinf′.
        gc_graph(γ′) ∗ tinf(tinf′) ∧
        v″ = newly_copied_vertex(γ, to) ∧
        γ′ = copy_vertex(γ, to, v′, v″) ∧
        compat(γ′, finf, tinf′, from, to) } ≜ ϕ27
28       Hd_val(v) = 0;
29       Field(v, 0) = (value)((void *)new);
30 // { ϕ27 ∧ val(hdr′) = 0 ∧ flds′[0] = copy(γ, v′) } ≜ ϕ30
31 //↘ { ∃flds. v ↦ flds ∧ flds = lab(γ′, v) } ≜ ϕ31
32       *p = (value)((void *)new);
33 //↙ { ϕ31 ∧ flds[0] := vaddr(γ, v″) }
34 // { ϕ30 ∧ ∃γ″. gc_graph(γ″) ∧
        γ″ = upd_edge(γ′, e, v″) ∧
        compat(γ″, finf, tinf′, from, to) }
35 }}}}}
36 // { fwd_postcondition(γ′, γ″, tinf′, finf, from, to, v, n) } ≜ ϕ36
```

$$fwd\_postcondition(\gamma, \gamma', tinf', finf', from, to, v, n) \triangleq \mathsf{gc\_graph}(\gamma') * \mathsf{finf}(finf') * \mathsf{tinf}(tinf') \wedge compat(\gamma', finf', tinf', from, to) \wedge forward\_relation(\gamma, \gamma', from, to, v, n)$$

Fig. 13. Clight code and proof sketch for forward

~~Lines~~ 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 ~~being~~ begin to show up in the C code, but the ~~verification looks~~ annotation on line 27 is relatively clean thanks to our mathematical graph framework~~. This~~: this is just the copying of a vertex, and so our new graph after the change is $\gamma' = copy\_vertex(\gamma, to, v', v'')$. 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 ~~done just~~ handled exactly as in line 15. The resultant graph is $\gamma'' = upd\_edge(\gamma', e, v'')$. 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 ~~and seen earlier, in that~~ `find` and `mark` seen earlier: it does not provide a relation saying that ~~has been functionally correct~~ `forward` has acted in a functionally "correct" manner. Rather, ~~we defined the relation to reflect the result of operations in , such as a vertex is~~ it uses *forward_relation* to carefully list the possible end results of calling `forward` on (v, n) — a vertex may be copied, an edge ~~is redirected, and etc, . For a taste, we~~ may

```
1  // { ∀γ, finf, tinf, from, to.
        gc_graph(γ) ∗ finf(finf) ∗ tinf(tinf) ∧
        compat(γ, finf, tinf, from, to) ∧
        st = start(γ, from) ∧ li = st + gensz(γ, from)
        ∧ sc = scan_start(γ, to) ∧ nx = naddr(tinf, to) } ≜ θ1
2  void do_scan(value *st, *li, *sc, **nx) {
3    value *s; s = sc;
4  // { θ1 ∧ s = scan_start(γ, to) } ≜ θ4
5    while(s < *nx) {
6  // { θ4 ∧ ∃n. scanned(γ, to) = s + n ∧
        scanned(γ, to) < naddr(tinf, to) } ≜ θ6
7  //↘ { ∃flds, hdr. flds = lab(scanned(γ, to)) ∧
        scanned(γ, to) ↦ flds ∧ hdr = flds[−1] } ≜ θ7
8      header_t hd = *((header_t *) s);
9  //↙ { θ7 ∧ hd = val(hdr) }
10 // { θ6 ∧ hd = val(hdr) } ≜ θ10
11     mlsize_t sz = Wosize_hd(hd);
12 // { θ10 ∧ sz = blocksize(hd) } ≜ θ12
13     int tag = Tag_hd(hd);
14     if (!No_scan(tag)) {
15       intnat j;
16       for(j = 1; j <= sz; j++) {
17 // { entails ϕ1 where v = s, n = j }
18         forward (st, li, nx, &Field(s, j));
19 // { entails ϕ36 where v = s, n = j
        i.e., ∃γ′, tinf′. gc_graph(γ′) ∧
        compat(γ′, finf, tinf′, from, to) ∧
        forward_relation(γ, γ′, from, to, s, j) } ≜ θ19
20       }
21     }
22     s += 1+sz;
23 }}
24 // { ds_postcondition(γ′, γ′, tinf′, finf, from, to) }
```

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

Fig. 14. Clight code and proof sketch for do_scan

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*~~in Appendix ??. We have relations of this sort~~ , with its twelve constructors, in the online version of our paper. We actually have such relations for all the key functions in our GC, and ~~we show that the~~ our final correctness proof shows that a systematic composition of these ~~correctly~~ relations corresponds to the ~~C~~ Clight code of our garbage collector. ~~The final functional correctness is derived from these relations.~~

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 `sc`anning. 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.

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. ~~Pointers and~~ Here the pointers `start` and `limit` point to the beginning and end of the $\texttt{i}^{\text{th}}$ space of the heap ~~.~~ `h`.

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

This subtraction is defined in C and Clight, but ~~will overflow~~ 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 ~~was~~ is unable to specify an OCaml-style GC such as ours. The first ~~involved~~ 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. ~~As mentioned in §6.3, needs to check~~ 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 ~~space. It uses this function :~~ `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 ~~and~~ `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 rather than returning .~~ 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 ~~because the comparison is bounded both above and below (in contrast, single-bounded comparisons need not be semantically preserved in CompCert).~~

~~The second 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 blockfields [Hickey et al. 2014]. To distinguish them, `forward` calls `Is_block` (line 5), which in turn calls When x is an int, this is indeed well-defined, but it is undefined to take the logical and of a pointer, so the code again gets stuck. Here the situation is messier, since CompCert does **not** guarantee that the alignment of a pointer is stable during compilation: in particular,~~ 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 ~~variables of type `char` may be packed tightly while assembling~~ variable of type `char`, which CompCert can realign as it assembles the stack frame~~, thus shifting their alignment. Informally, of course, this cannot occur in the GC since we do not store stack-allocated local variables in the GC-managed heap. We have discussed this issue with the CompCert team [Leroy 2018], and believe that a stronger specification of the extcall_properties (essentially, guaranteeing that non-`char` pointers do not change alignment) should allow us to prove that CompCertwill respect `test_int_or_ptr`'s behavior. The CompCert team understands the issue [?]and wants OCaml-style GCs to have defined behavior in Clight.~~. 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 6) were~~ 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)~~ 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 ~~Floyd~~ 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 ~~into Floyd~~ (as described in §2–§3) into Floyd.

7.2 ~~Localizations in VST with and~~

~~Floyd~~ The Floyd modulepresents 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 ~~will then automatically apply~~ 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 proof is finished from the top to "the current program point")

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`

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 ~~four~~ 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 ~~saw~~ see the assertion P_2 (line 3) and ~~decided~~ decide to use the `localize` tactic to zoom into P_3 (line 4). They then ~~applied~~ 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 $P_5 \twoheadrightarrow P_6$. In truth, $?F$ is chosen more subtly to properly handle both existential variables and modified program variables~~;~~. Then `unlocalize` ~~then~~ automatically simplifies the goals to present a cleaner interface to the user. These transformations require the additional theory given in §3.

7.2 ~~Statistics related to~~ Modularity of our ~~development~~Library

We worked to make our library modular, thus encouraging proof reuse. Figure 16 gives a sense of this engineering effort. 14 files were used to verify the garbage collector, and the bar at the bottom shows their relative lengths. Each straight line in the perimeter represents a theorem used to verify the GC, and the length of a line corresponds to the length of the theorem on a logarithmic scale. The 14 color-coded sectors represent the files in which these theorems are housed. Arcs connecting theorems represent dependencies, where an arc is colored the same as the theorem's *caller*.

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

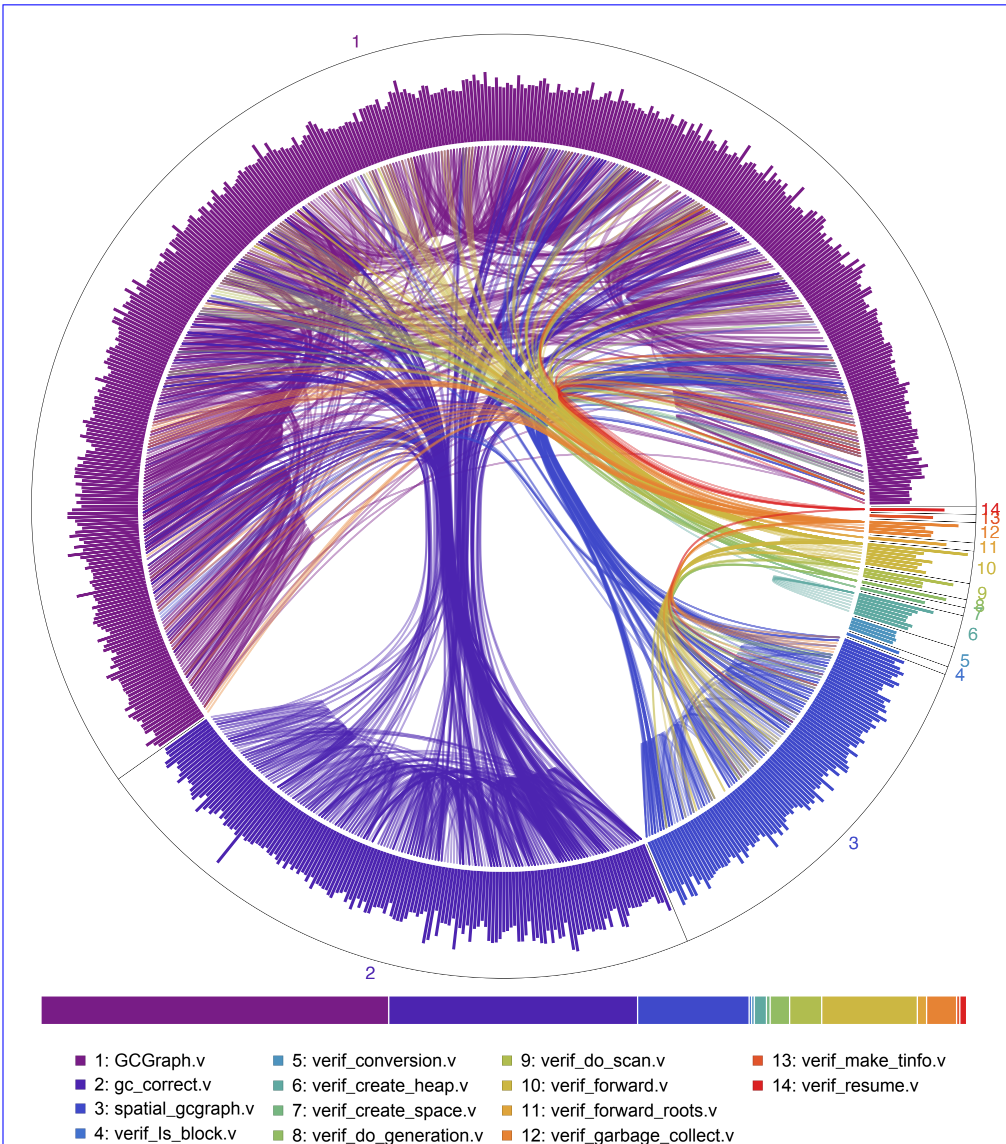

Fig. 16. Theorems in the verification of the GC

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.

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 ~~with~~ that development's importance or ~~implementation difficulty~~ the difficulty of its implementation, we present the size nonetheless in Table 1. Our proof script is 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. For context, 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 us eight months.

8 RELATED WORK

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

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

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~~Figure 3. ~~Hobor and Villard~~ 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.

~~Our development is entirely machine-checked (§7) which revealed some tricky technique details.~~ Hobor and Villard ~~fell into the trap of defining spatial graphs recursively (§5.2); unfortunately other members of the research community have since followed them in. We exposed this error~~

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

~~Statistics for our code base~~

~~and provided a sound, general, and highly modular graph framework that works smoothly in a mechanized context (§4,§5).~~

~~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 ~~very modular and general and~~ 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 ~~(§5.2); unfortunately~~. Unfortunately, other members of the research community ~~have since followed them in,~~ (*e.g.* ~~[Raad et al. 2015]. We exposed this errorand provided~~ Raad et al. [2015]) followed their lead. We expose this error (§5.2) and provide a sound and ~~quite~~ rather general definition for graph ~~(§5.3)~~ that recovers fold/unfold reasoning~~. We developed~~ (§5.3). We develop a much more general and more modular set of related lemmas and ~~connect~~ connected our spatial reasoning to the verification framework of CompCert/VST (§7)~~, and our~~. Our development is entirely machine-checked (§7) whereas they used only pen and paper.

Other Pen-and-Paper Verification of Graph Algorithms and/or $\cup\!\!\!*$.

~~*Other verification of graph algorithms and/or $\cup\!\!\!*$.*~~ ~~[Yang 2001]'s verification of~~ Yang [2001] verified the Schorr-Waite algorithm~~is~~, and this is widely considered a landmark in the early separation logic literature. ~~[Krishnaswami 2011] provided the first separation logic proof of union-find.~~ ~~[Bornat et al. 2004]~~ 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]~~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]~~Gardner et al. [2012] were the first to reason about a program using $\cup\!\!\!*$ in Javascript. ~~[Raad et al. 2015]~~Raad et al. [2015] used $\cup\!\!\!*$ within their CoLoSL program logic to reason about a concurrent spanning algorithm using a kind of "concurrent localization". ~~[Sergey et al. 2015] also verified~~

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.

~~A decade after Yang verified Schorr-Waite on paper, [Leino 2010] automated its verification.~~ 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 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 [2015, 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~~Tools in Coq. Our work interacts with the Floyd ~~[Appel et al. 2014] verification tool.~~ 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]. ~~A~~

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 [2015, 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 ~~Coq is the Bedrock framework [Chlipala 2011].~~ 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.

~~Many automated~~

Other Verification Tools. Many more-automated verification tools also use separation logic in a forward reasoning style~~as does HIP/SLEEK, including~~ . Smallfoot [Berdine et al. 2005], jStar ~~[Distefano and Park~~ ~~, and Verifast [Jacobs et al. 2011]. One of~~ [Distefano and Parkinson 2008], HIP/SLEEK~~'s distinguishing features is good support for user-defined inductive predicates rather than a library of pre-defined predicates for lists, trees etc.~~

~~Dafny [Leino 2010] and~~ [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] ~~are verifiers~~ 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~~Mathematical Graph Theory. There is a long history, going back at least ~~25~~ 28 years, of mechanized reasoning about mathematical graphs [Wong 1991]. The most famous mechanically verified ~~"graph theorem"~~ "graph theorem" is the Four Color Theorem [Gonthier 2005]; however the development actually uses hypermaps instead of graphs. ~~Noschinski~~ In general most "mathematical graph" frameworks in the literature [Butler and Sjogren 1998; Chou 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 ~~[Noschinski 2015b]~~, *e.g.* supporting graphs with labeled and parallel arcs. ~~[Dubois et al. 2015; Noschinski 2015a] used~~ 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; Yamamoto et al. 1995]~~ ~~use~~ Bauer and Nipkow [2002] and Yamamoto et al. [1995] used an inductive encoding of graphs to formalize planar graph theory.

~~[Guéneau et al. 2018] formalised "time credits" in Separation Logic and big-O notation in Coq and then verified both the correctness and the time complexity of the union-find data structure.~~

Verification of ~~garbage collection algorithms~~Garbage Collection Algorithms. Schism [Gammie et al. 2015; Pizlo et al. 2010] is a certified concurrent collector built in a Java VM that services multicore architectures with weak memory consistency. ~~McCreight *et al.* [McCreight et al. 2010, 2007] introduce~~ 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. ~~Hawblitzel and Petrank [Petrank and Hawblitzel 2010] annotate~~ Petrank and Hawblitzel [2010] annotated x86 code for two GCs by hand, and then ~~use~~ 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, ~~in particular to add~~ 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 ~~is~~ 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 and the National Science Foundatio grant CCF-1521602. 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 or the National Science Foundation.

10 ~~SPANNING AND COPYING~~

```
struct Node {
    int m;
    struct Node * l;
    struct Node * r; };
// We use R to represent reachable(γ, x)
void spanning(struct Node * x) {
// {graph(x, γ) ∧ γ(x).1 = 0}
    struct Node * l, * r; int root_mark;
// {graph(x, γ) ∧ ∃l, r. γ(x) = (0, l, r)}
// {graph(x, γ) ∧ γ(x) = (0, l, r)}
// {vertices_at(reachable(γ, x), γ) ∧ γ(x) = (0, l, r)}
// {vertices_at(R, γ) ∧ γ(x) = (0, l, r)}
// ↘ {x ↦ 0, l, r ∧ γ(x) = (0, l, r)}
    l = x -> l; r = x -> r; x -> m = 1;
// ↙ {x ↦ 1, l, r ∧ γ(x) = (0, l, r) ∧ ∃γ1. mark1(γ, x, γ1)}
// {∃γ1. vertices_at(R, γ1) ∧ γ(x) = (0, l, r) ∧ mark1(γ, x, γ1)}
// {vertices_at(R, γ1) ∧ γ(x) = (0, l, r) ∧ mark1(γ, x, γ1)}
    if (l) {
```

```
        root_mark = l -> m;
        if (root_mark == 0) {
            spanning(l);
        } else { x -> l = 0; } }
```

// $\left\{\begin{array}{l}\exists \gamma_2.\ \mathsf{vertices\_at}(R, \gamma_2) \land \gamma(\mathrm{x}) = (0, \mathrm{l}, \mathrm{r}) \land \\ \mathit{mark1}(\gamma, \mathrm{x}, \gamma_1) \land \mathit{e\_span}(\gamma_1, \mathrm{x.L}, \gamma_2)\end{array}\right\}$

// $\left\{\begin{array}{l}\mathsf{vertices\_at}(R, \gamma_2) \land \gamma(\mathrm{x}) = (0, \mathrm{l}, \mathrm{r}) \land \\ \mathit{mark1}(\gamma, \mathrm{x}, \gamma_1) \land \mathit{e\_span}(\gamma_1, \mathrm{x.L}, \gamma_2)\end{array}\right\}$

```
    if (r) {
        root_mark = r -> m;
        if (root_mark == 0) {
            spanning(r);
        } else { x -> r = 0; } }
```

// $\left\{\begin{array}{l}\exists \gamma_3.\ \mathsf{vertices\_at}(R, \gamma_3) \land \gamma(\mathrm{x}) = (0, \mathrm{l}, \mathrm{r}) \land \\ \mathit{mark1}(\gamma, \mathrm{x}, \gamma_1) \land \mathit{e\_span}(\gamma_1, \mathrm{x.L}, \gamma_2) \land \mathit{e\_span}(\gamma_2, \mathrm{x.R}, \gamma_3)\end{array}\right\}$

`}` // $\{\exists \gamma_3.\ \mathsf{vertex\_at}(\mathsf{reachable}(\gamma, \mathrm{x}), \gamma_3) \land \mathit{span}(\gamma, \mathrm{x}, \gamma_3)\}$

$$\mathsf{vertices\_at}(\mathsf{reachable}(\gamma_1, \mathrm{x}), \gamma_2) \triangleq \mathop{\bigstar}_{v \in \mathsf{reachable}(\gamma_1, \mathrm{x})} v \mapsto \gamma_2(v)$$

$$\begin{aligned}\mathit{span}(\gamma_1, \mathrm{x}, \gamma_2) \triangleq\ & \mathit{mark}(\gamma_1, \mathrm{x}, \gamma_2) \land \gamma_1 \uparrow (\lambda v. x \overset{\gamma_1}{\rightsquigarrow}{}_0^{\star}\, v) \text{ is a tree} \land \\ & \gamma_1 \uparrow (\lambda v. \neg x \overset{\gamma_1}{\rightsquigarrow}{}_0^{\star}\, v) = \gamma_2 \uparrow (\lambda v. \neg x \overset{\gamma_1}{\rightsquigarrow}{}_0^{\star}\, v) \land \\ & (\forall v.\ x \overset{\gamma_1}{\rightsquigarrow}{}_0^{\star}\, v \Rightarrow \gamma_2 \models x \rightsquigarrow v) \land \\ & (\forall a, b.\ x \overset{\gamma_1}{\rightsquigarrow}{}_0^{\star}\, a \Rightarrow \neg x \overset{\gamma_1}{\rightsquigarrow}{}_0^{\star}\, b \Rightarrow \neg \gamma_2 \models a \rightsquigarrow b)\end{aligned}$$

$$\mathit{e\_span}(\gamma_1, e, \gamma_2) \triangleq \begin{cases} \gamma_1 - e = \gamma_2 & t(\gamma_1, e) = 1 \\ \mathit{span}(\gamma_1, t(\gamma_1, e), \gamma_2) & t(\gamma_1, e) = 0 \end{cases}$$

~~Clight code and proof sketch for bigraph spanning tree.~~

~~In Figure ?? we show a simplified proof script for the spanning tree algorithm. Unlike graph marking, the spanning tree algorithm changes the structure of the graph, leading to a more complicated specification, in both the pure part and the spatial part. Observe that the *span* relation is rather long; the *e_span* handles the case of either calling spanning tree or deleting an edge.~~

~~We put the proof sketch of the graph copying algorithm in Figure ?? and Figure ??. Just like other parts of the paper, both algorithms have been machine verified.~~

```
struct Node {
    int m;
    struct Node * l;
    struct Node * r; };
```

// We use $x \overset{\gamma}{\longleftrightarrow} x'$ to represent $x = x' = 0 \lor \gamma(x) = (x', \_, \_)$

```
struct Node * copy(struct Node * x) {
    struct Node * l, * r, * x0, * l0, * r0;
```

// $\{\mathsf{graph}(\mathrm{x}, \gamma)\}$

```
      if (x == 0)
        return 0;
```

// $\{\mathsf{graph}(\mathrm{x}, \gamma) \land x \neq 0\}$

```
// $\{graph(x, \gamma) \land \exists x_0, l, r.\ \gamma(x) = (x_0, l, r)\}$
// $\{graph(x, \gamma) \land \gamma(x) = (x_0, l, r)\}$
// $\searrow \{x \mapsto x_0, l, r \land \gamma(x) = (x_0, l, r)\}$
      x0 = x -> m;
// $\swarrow \{x \mapsto x_0, l, r \land \gamma(x) = (x_0, l, r) \land x0 = x_0\}$
// $\{graph(x, \gamma) \land \gamma(x) = (x0, l, r)\}$
      if (x0 != 0)
        return x0;
// $\{graph(x, \gamma) \land \gamma(x) = (0, l, r)\}$
      x0 = (struct Node *) mallocN (sizeof (struct Node));
// $\{graph(x, \gamma) * x0 \mapsto \_, \_, \_ \land \gamma(x) = (0, l, r)\}$
// $\searrow \{x \mapsto 0, l, r * x0 \mapsto \_, \_, \_ \land \gamma(x) = (0, l, r)\}$
      l = x -> l; r = x -> r; x -> m = x0; x0 -> m = 0;
// $\swarrow \left\{\begin{array}{l} x \mapsto x0, l, r * x0 \mapsto 0, \_, \_ \land \\ \gamma(x) = (0, l, r) \land \exists \gamma_1 \gamma_1'.\ v\_copy1(\gamma, x, \gamma_1, \gamma_1') \end{array}\right\}$
// $\left\{\begin{array}{l} \exists \gamma_1 \gamma_1'.graph(x, \gamma_1) * x0 \mapsto 0, \_, \_ \land \\ \gamma(x) = (0, l, r) \land v\_copy1(\gamma, x, \gamma_1, \gamma_1') \end{array}\right\}$
// $\left\{\begin{array}{l} graph(x, \gamma_1) * x0 \mapsto 0, \_, \_ \land \\ \gamma(x) = (0, l, r) \land v\_copy1(\gamma, x, \gamma_1, \gamma_1') \end{array}\right\}$
// $\left\{\begin{array}{l} graph(x, \gamma_1) * x0 \mapsto 0, \_, \_ * holegraph(x0, \gamma_1') \land \\ \gamma(x) = (0, l, r) \land v\_copy1(\gamma, x, \gamma_1, \gamma_1') \end{array}\right\}$
// $\searrow \{graph(l, \gamma_1)\}$
      l0 = copy(l);
// $\swarrow \left\{\begin{array}{l} \exists \gamma_2 \gamma_2''.graph(l, \gamma_2) * graph(l0, \gamma_2'') \land \\ copy(\gamma_1, l, \gamma_2, \gamma_2'') \land l \overset{\gamma_2}{\longleftrightarrow} l0 \end{array}\right\}$
// $\left\{\begin{array}{l} \exists \gamma_2 \gamma_2''.graph(x, \gamma_2) * x0 \mapsto 0, \_, \_ * holegraph(x0, \gamma_1') * \\ graph(l0, \gamma_2'') \land \gamma(x) = (0, l, r) \land v\_copy1(\gamma, x, \gamma_1, \gamma_1') \land \\ copy(\gamma_1, l, \gamma_2, \gamma_2'') \land l \overset{\gamma_2}{\longleftrightarrow} l0 \end{array}\right\}$
// $\left\{\begin{array}{l} \exists \gamma_2 \gamma_2'.graph(x, \gamma_2) * x0 \mapsto 0, \_, \_ * holegraph(x0, \gamma_2') \land \\ \gamma(x) = (0, l, r) \land v\_copy1(\gamma, x, \gamma_1, \gamma_1') \land \\ e\_copy(\gamma_1, \gamma_1', x.L, \gamma_2, \gamma_2') \land l \overset{\gamma_2}{\longleftrightarrow} l0 \end{array}\right\}$
// $\left\{\begin{array}{l} graph(x, \gamma_2) * x0 \mapsto 0, \_, \_ * holegraph(x0, \gamma_2') \land \\ \gamma(x) = (0, l, r) \land v\_copy1(\gamma, x, \gamma_1, \gamma_1') \land \\ e\_copy(\gamma_1, \gamma_1', x.L, \gamma_2, \gamma_2') \land l \overset{\gamma_2}{\longleftrightarrow} l0 \end{array}\right\}$
      x0 -> l = l0;
// $\left\{\begin{array}{l} graph(x, \gamma_2) * x0 \mapsto 0, l0, \_ * holegraph(x0, \gamma_2') \land \\ \gamma(x) = (0, l, r) \land v\_copy1(\gamma, x, \gamma_1, \gamma_1') \land \\ e\_copy(\gamma_1, \gamma_1', x.L, \gamma_2, \gamma_2') \land l \overset{\gamma_2}{\longleftrightarrow} l0 \end{array}\right\}$
// $\searrow \{graph(r, \gamma_2)\}$
      r0 = copy(r);
// $\swarrow \left\{\begin{array}{l} \exists \gamma_3 \gamma_3''.graph(r, \gamma_3) * graph(r0, \gamma_3'') \land \\ copy(\gamma_2, r, \gamma_3, \gamma_3'') \land r \overset{\gamma_3}{\longleftrightarrow} r0 \end{array}\right\}$
// $\left\{\begin{array}{l} \exists \gamma_3 \gamma_3''.graph(x, \gamma_3) * x0 \mapsto 0, l0, \_ * holegraph(x0, \gamma_3') * \\ graph(r0, \gamma_3'') \land \gamma(x) = (0, l, r) \land v\_copy1(\gamma, x, \gamma_1, \gamma_1') \land \\ e\_copy(\gamma_1, \gamma_1', x.L, \gamma_2, \gamma_2') \land copy(\gamma_2, r, \gamma_3, \gamma_3'') \land \\ l \overset{\gamma_2}{\longleftrightarrow} l0 \land r \overset{\gamma_3}{\longleftrightarrow} r0 \end{array}\right\}$
// $\left\{\begin{array}{l} \exists \gamma_3 \gamma_3'.graph(x, \gamma_3) * x0 \mapsto 0, l0, \_ * holegraph(x0, \gamma_3') \land \\ \gamma(x) = (0, l, r) \land v\_copy1(\gamma, x, \gamma_1, \gamma_1') \land \\ e\_copy(\gamma_1, \gamma_1', x.L, \gamma_2, \gamma_2') \land e\_copy(\gamma_2, \gamma_2', x.R, \gamma_3, \gamma_3') \land \\ l \overset{\gamma_2}{\longleftrightarrow} l0 \land r \overset{\gamma_3}{\longleftrightarrow} r0 \end{array}\right\}$
```

~~Proof sketch for bigraph copy – part 1~~

$$// \left\{\begin{array}{l}\text{graph}(\mathtt{x}, \gamma_3) * \mathtt{x0} \mapsto 0, \mathtt{l0}, \_ * \text{holegraph}(\mathtt{x0}, \gamma_3') \wedge \\ \gamma(\mathtt{x}) = (0, \mathtt{l}, \mathtt{r}) \wedge \mathit{v\_copy1}(\gamma, \mathtt{x}, \gamma_1, \gamma_1') \wedge \\ \mathit{e\_copy}(\gamma_1, \gamma_1', \mathtt{x}.L, \gamma_2, \gamma_2') \wedge \mathit{e\_copy}(\gamma_2, \gamma_2', \mathtt{x}.R, \gamma_3, \gamma_3') \wedge \\ \mathtt{l} \overset{\gamma_2}{\longleftrightarrow} \mathtt{l0} \wedge \mathtt{r} \overset{\gamma_3}{\longleftrightarrow} \mathtt{r0}\end{array}\right\}$$

```
    x0 -> r = r0;
```

$$// \left\{\begin{array}{l}\text{graph}(\mathtt{x}, \gamma_3) * \mathtt{x0} \mapsto 0, \mathtt{l0}, \mathtt{r0} * \text{holegraph}(\mathtt{x0}, \gamma_3') \wedge \\ \gamma(\mathtt{x}) = (0, \mathtt{l}, \mathtt{r}) \wedge \mathit{v\_copy1}(\gamma, \mathtt{x}, \gamma_1, \gamma_1') \wedge \\ \mathit{e\_copy}(\gamma_1, \gamma_1', \mathtt{x}.L, \gamma_2, \gamma_2') \wedge \mathit{e\_copy}(\gamma_2, \gamma_2', \mathtt{x}.R, \gamma_3, \gamma_3') \wedge \\ \mathtt{l} \overset{\gamma_2}{\longleftrightarrow} \mathtt{l0} \wedge \mathtt{r} \overset{\gamma_3}{\longleftrightarrow} \mathtt{r0}\end{array}\right\}$$

$$// \left\{\text{graph}(\mathtt{x}, \gamma_3) * \text{graph}(\mathtt{x0}, \gamma_3') \wedge \mathit{copy}(\gamma, \mathtt{x}, \gamma_3, \gamma_3') \wedge \mathtt{x} \overset{\gamma_3}{\longleftrightarrow} \mathtt{x0}\right\}$$

$$\text{holegraph}(x, \gamma) \triangleq \mathop{\Huge *}_{v \in \text{reachable}(\gamma, x) - \{x\}} v \mapsto \gamma(v)$$

$$\begin{aligned} \mathit{iso}(f_V, f_E, \gamma_1, \gamma_2) \triangleq\ & f_V \text{ is a bijection between } \phi_V(\gamma_1) \text{ and } \phi_V(\gamma_2) \wedge \\ & f_E \text{ is a bijection between } \phi_E(\gamma_1) \text{ and } \phi_E(\gamma_2) \wedge \\ & \forall e, f_V(s(\gamma_1, e)) = s(\gamma_2, f_E(e)) \wedge \\ & \forall e, f_V(d(\gamma_1, e)) = d(\gamma_2, f_E(e)) \end{aligned}$$

$$\begin{aligned} \mathit{v\_copy1}(\gamma_1, x, \gamma_2, \gamma_2') \triangleq\ & \exists x'. x \neq 0 \wedge \mathit{mark1}(\gamma_1, x, \gamma_2) \wedge \\ & x \overset{\gamma_2}{\longleftrightarrow} x' \wedge \gamma_2' = \{x_0\} \end{aligned}$$

$$\begin{aligned} \mathit{copy}(\gamma_1, x, \gamma_2, \gamma_2') \triangleq\ & \mathit{mark}(\gamma_1, x, \gamma_2) \wedge \\ & \exists f_V f_E. \mathit{iso}(f_V, f_E, \gamma_2 \uparrow (\lambda v. x \overset{\gamma_2}{\rightsquigarrow}{}_0^{\star} v), \gamma_2') \wedge \\ & \forall x x'. f_V(x) = x' \Leftrightarrow x \overset{\gamma_2}{\longleftrightarrow} x' \end{aligned}$$

$$\begin{aligned} \mathit{e\_copy}(\gamma_1, \gamma_1', e, \gamma_2, \gamma_2') \triangleq\ & \exists \gamma_2''. \gamma_2' = \gamma_1' + \gamma_2'' \wedge \\ & \mathit{mark}(\gamma_1, x, \gamma_2) \wedge \exists f_V f_E. \\ & \mathit{iso}(f_V, f_E, \{e\} + \gamma_2 \uparrow (\lambda v. x \overset{\gamma_2}{\rightsquigarrow}{}_0^{\star} v), \gamma_2'') \wedge \\ & \forall x x'. f_V(x) = x' \Leftrightarrow x \overset{\gamma_2}{\longleftrightarrow} x' \end{aligned}$$

Here, when we mention *mark* and *mark1*, the value 1s in the original definition are changed to non-zero values.

~~Proof sketch for bigraph copy - part 2~~

10 ~~DIFFICULTY USING GRAPH_T~~

$$\dfrac{\dfrac{100 \mapsto 42, 100, 0 \ \vdash\ 100 \mapsto 42, 100, 0 \uplus \text{graph}_T(100, \hat{\gamma})}{100 \mapsto 42, 100, 0 \ \vdash\ \hat{\gamma}(100) = (42, 100, 0) \ \wedge\ 100 \mapsto 42, 100, 0 \uplus \text{graph}_T(100, \hat{\gamma}) \uplus \text{graph}_T(0, \hat{\gamma})}\ (2)}{100 \mapsto 42, 100, 0 \ \vdash\ \text{graph}_T(100, \hat{\gamma})}\ (1)$$

~~(1) Unfold graph_T, dismiss first disjunct (contradiction), introduce existentials (which must be 42,100,0) (2) simplify using $P * \mathsf{emp} \dashv\vdash P$ and remove pure conjunct~~

~~An attempt to prove a "simple" entailment~~

~~See Figure 11 for an attempt to prove the entailment $100 \mapsto 42, 100, 0 \vdash \mathsf{graph}_T(100, \hat{\gamma})$. 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.~~

10 ~~PROBLEM WITH APPEL AND MCALLESTER'S FIXPOINT~~

~~Appeland 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 the "approximation modality" $\triangleright$ [Appel et al. 2007], *i.e.* our graph predicate would look like~~

$$\begin{aligned}\underline{\mathsf{graph}_A(x,\gamma)} &\quad \underline{\triangleq} \\ \underline{(x = 0 \land \mathsf{emp})} &\quad \underline{\lor \exists m,l,r.\ \gamma(x) = (m,l,r) \land} \\ \underline{x \mapsto m,l,r} &\quad \underline{\uplus \triangleright \mathsf{graph}_A(l,\gamma) \uplus \triangleright \mathsf{graph}_A(r,\gamma)}\end{aligned}$$

~~Unfortunately, $\triangleright P$ is not precise for all P, so graph_A is not precise either. The approximation modality's universal imprecision has never been noticed before.~~

10 ~~STRUCTURE OF THE GARBAGE COLLECTOR PROGRAM~~

~~CertiCoq uses a generational copying garbage collector that is inspired by the OCaml GC. The heap is divided into a series of disjoint spaces called *generations*. The size of the first generation is carefully calculated, and then subsequent generations double in size. The mutator only ever allocates new memory in the first, smallest generation of heap, which is called the nursery. If it finds that the nursery is full, the mutator calls the GC to free up space. The GC collects the nursery (now called the *from* generation) into the second generation (the *to* generation): it examines the elements in *from*, sees if they are accessible by the mutator, and, if they are, copies them over to *to*. This copying is achieved over a few steps, and we will explain these shortly, but the larger picture is that everything of importance in *from* gets copied to *to*, and so *from* can safely be reset.~~

~~An important subtlety here is that *to* had enough room to accept *from*'s items. In the (empirically improbable) worst case, *all* of *from*'s fields were copied over to *to*. Because *to* has twice the capacity of *from*, *to* could not have been more than half full when the collection started. This guarantee must be renewed before the next collection. So, in case the collection of the nursery caused the second generation to become more than half full, the second generation is collected into the third. This makes both the first and second generations empty, thus ensuring the guarantee trivially. It should be clear to see that this may also trigger further collections in a cascade effect. The GC's task is only complete once this cascade (if any) is over. It returns control to the mutator, which goes ahead with the allocation that it was trying to perform in the nursery.~~

~~Having shown that the overall collection works via (a series of) two-generational collections, we now zoom in and explain a two-generation collection. The GC starts at the mutator-owned arguments array, whose fields are either data, or pointers that point at memory blocks in the heap. It ignores the data entirely, and, among the pointers, cares only for the pointers that point into the *from* generation. For each pointer that points into *from*, it copies its target block to *to*, simply adding it in its entirety after *to*'s last-used memory field, which is called *next*.~~

~~This operation only takes care of the blocks in the heap that the arguments array was pointing at directly, so the GC still has to copy over indirectly-accessible blocks. Of course, the only way to access an indirect block is via one of the direct blocks that it has just finished copying into a contiguous array. It starts at the old *next* in *to* and works its way "upwards" through the freshly copied blocks, again looking exclusively for pointers that point into *from* and copying over their target blocks into *to*. In the *to* generation, these newly copied blocks simply get stacked atop our first batch of copied blocks.~~

~~The mutator's dependency graph has indefinite depth, so the second batch of copied blocks may still have pointers into *from*. However, thanks to this systematic copying strategy, it is very easy to take care of all indirect blocks. The GC simply keeps scanning upwards in *to*, copying over blocks from *from* as necessary, until the scanning pointer catches up to the last-used field in *to*. This completes a collection from *from* to *to*, copying all blocks that lived in *from* and were of interest to the mutator. *from* is now reset.~~

~~A good question at this juncture is why this rather selective scan of the args array and the heap is good enough to collect *from*. The GC definitely collected every direct block by scanning the args array, but what of the indirect blocks? Couldn't there be valid indirect links that start either below *next* in the *to* generation, or from other generations altogether?~~

~~Both of these turn out to be impossible because the mutator behaves in a purely functional manner. The heap is chronologically faithful, in that higher-indexed generations host objects that were allocated earlier. Because of the immutability of objects in a purely functional language, it is impossible for objects to point "backwards" to a younger generation, as the older object would not have had known about the younger at the time of its allocation, and could not have been modified after its allocation. Fields living in generations younger than *from* can point into *from* during normal mutator activity, but this is impossible at the time of collection: *from* is only ever collected either if it is the nursery or if a cascade effect has caused all generations younger than it to be collected and reset. In fact, the only time the GC ever sees backwards pointers is when it creates (and quickly fixes) them during during its activities.~~

10 ~~CODE FOR FORWARD RELATION~~

```
  Inductive
forward_relation (from to : nat) : nat -> forward_t -> LGraph -> LGraph -> Prop :=
    fr_z : forall (depth : nat) (z : Z) (g : LGraph),
           forward_relation from to depth (inl (inl (inl z))) g g
  | fr_p : forall (depth : nat) (p : GC_Pointer) (g : LGraph),
           forward_relation from to depth (inl (inl (inr p))) g g
  | fr_v_not_in : forall (depth : nat) (v : VType) (g : LGraph),
                  vgeneration v <> from ->
                  forward_relation from to depth (inl (inr v)) g g
  | fr_v_in_forwarded : forall (depth : nat) (v : VType)
                          (g : LabeledGraph VType EType raw_vertex_block unit
                                 graph_info),
                        vgeneration v = from ->
                        raw_mark (vlabel g v) = true ->
                        forward_relation from to depth (inl (inr v)) g g
  | fr_v_in_not_forwarded_O : forall (v : VType)
                                (g : LabeledGraph VType EType raw_vertex_block unit
                                       graph_info),
                              vgeneration v = from ->
                              raw_mark (vlabel g v) = false ->
                              forward_relation from to 0
                                (inl (inr v)) g (lgraph_copy_v g v to)
  | fr_v_in_not_forwarded_Sn : forall (depth : nat) (v : VType)
                                 (g : LabeledGraph VType EType raw_vertex_block unit
                                        graph_info) (g' : LGraph),
                               vgeneration v = from ->
                               raw_mark (vlabel g v) = false ->
                               let new_g := lgraph_copy_v g v to in
```

```
                                 forward_loop from to depth
                                   (vertex_pos_pairs new_g (new_copied_v g to)) new_g
                                   g' ->
                                 forward_relation from to (S depth) (inl (inr v)) g g'
  | fr_e_not_to : forall (depth : nat) (e : EType) (g : LGraph),
                  vgeneration (dst (pg_lg g) e) <> from ->
                  forward_relation from to depth (inr e) g g
  | fr_e_to_forwarded : forall (depth : nat) (e : EType) (g : LGraph),
                        vgeneration (dst (pg_lg g) e) = from ->
                        raw_mark (vlabel g (dst (pg_lg g) e)) = true ->
                        let new_g :=
                          labeledgraph_gen_dst g e
                            (copied_vertex (vlabel g (dst (pg_lg g) e))) in
                        forward_relation from to depth (inr e) g new_g
  | fr_e_to_not_forwarded_O : forall (e : EType) (g : LGraph),
                              vgeneration (dst (pg_lg g) e) = from ->
                              raw_mark (vlabel g (dst (pg_lg g) e)) = false ->
                              let new_g :=
                                labeledgraph_gen_dst
                                  (lgraph_copy_v g (dst (pg_lg g) e) to) e
                                  (new_copied_v g to) in
                              forward_relation from to 0 (inr e) g new_g
  | fr_e_to_not_forwarded_Sn : forall (depth : nat) (e : EType) (g g' : LGraph),
                               vgeneration (dst (pg_lg g) e) = from ->
                               raw_mark (vlabel g (dst (pg_lg g) e)) = false ->
                               let new_g :=
                                 labeledgraph_gen_dst
                                   (lgraph_copy_v g (dst (pg_lg g) e) to) e
                                   (new_copied_v g to) in
                               forward_loop from to depth
                                 (vertex_pos_pairs new_g (new_copied_v g to)) new_g
                                 g' ->
                               forward_relation from to (S depth) (inr e) g g'
  with forward_loop (from to : nat)
         : nat -> list forward_p_type -> LGraph -> LGraph -> Prop :=
    fl_nil : forall (depth : nat) (g : LGraph), forward_loop from to depth [] g g
  | fl_cons : forall (depth : nat) (g1 g2 g3 : LGraph) (f : forward_p_type)
                (fl : list forward_p_type),
              forward_relation from to depth (forward_p2forward_t f [] g1) g1 g2 ->
              forward_loop from to depth fl g2 g3 ->
              forward_loop from to depth (f :: fl) g1 g3
```

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. 2015. Machine-Checked Verification of the Correctness and Amortized Complexity of an Efficient Union-Find Implementation. In *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings*. 137–153. https://doi.org/10.1007/978-3-319-22102-1_9

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, Arthur Charguéraud, and François Pottier. 2018. A Fistful of Dollars: Formalizing Asymptotic Complexity Claims via Deductive Program Verification. In *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*. 533–560. https://doi.org/10.1007/978-3-319-89884-1_19

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.

Xavier Leroy. 2018. Re: The dark corners of C light pointer semantics.

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.