The Ramifications of Mechanized Localizations within Data Structures

Shengyi Wang* Qinxiang Cao+ Asankhaya Sharma* Aquinas Hobor†,*

School of Computing* and Yale-NUS College†, National University of Singapore; Princeton University+

Abstract

We develop a way to mechanically verify realistic programs that manipulate data structures with intrinsic sharing such as heap-represented graphs. We upgrade Hobor and Villard's theory of ramification to better support modified program variables and existential quantifiers in assertions. We develop a modular and general setup for reasoning about mathematical graphs and show how to connect this setup to a general theory of graphs in separation logic. We connect our development to two large verification tools, the Verified Software Toolchain and HIP/SLEEK, and use these tools to mechanically verify several canonical graph algorithms.

1. Introduction

Over the last fifteen years great strides have been made in automating verifications of programs that manipulate tree-like data structures using separation logic (Berdine et al. 2005; Chin et al. 2010; Jacobs et al. 2011; Chlipala 2011; Bengtson et al. 2012; Appel et al. 2014). Unfortunately, verifying programs that manipulate graph-like data structures (i.e. structures with *intrinsic sharing*) has been more challenging. Indeed, verifying such programs was formidable enough that a number of the early landmark results in separation logic devoted substantial effort to verify single examples such as Schorr-Waite (Yang 2001) with pen and paper—avoiding the additional challenges inherent in mechanized reasoning.

In recent years, Hobor and Villard introduced the concept of *ramification* as a kind of proof pattern or framework to verify graph-manipulating programs on pen and paper (Hobor and Villard 2013). The major focus of this paper is to develop methods to verify realistic graph programs in a mechanized context. We do so by upgrading the theory of ramification and by developing a general and modular library for graph-related reasoning in separation logic. We incorporate our approach into two sizeable separation logic-based verification tools: the Floyd system of the Verified Software Toolchain (VST) (Appel et al. 2014) and the HIP/SLEEK program verifier (Chin et al. 2010). VST and HIP/SLEEK inhabit quite different points in the design space for verification tools, with VST primarily focused on heavily human-guided verifications with an emphasis on end-to-end machine-checked proofs, and HIP/SLEEK focusing on more automation. Despite these differences, the vast majority of our Coq code base is shared between them, giving us hope that our work will be applicable to other verification tools.

The structure of our paper is as follows:

§2 We verify a graph marking algorithm and explain why such algorithms are easier to verify using relations instead of functions. We introduce *localization blocks* as a new notation for ramification. We upgrade Hobor and Villard's RAMIFY rule to handle both modified program variables and existential quantifiers more gracefully.

§3 We develop a general mechanization of mathematical graphs powerful enough to support realistic verification.

§4 We suggest that the standard 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 way that can be utilized in multiple flavors of separation logic, such as the logics contained in VST and HIP/SLEEK.

§5 We explain how we integrated ramification into VST by developing two new Floyd tactics, `localize` and `unlocalize`. We discuss three additional VST-certified examples: marking a DAG, pruning a graph into a spanning tree (*e.g.* for disposal), and making a structure-preserving copy of a graph.

§6 We explain how we modified HIP/SLEEK to introduce ramifications when programs modify data structures with intrinsic sharing and to automatically discharge the associated obligations using Coq-verified external lemmas.

§7 We document some statistics related to our development.

§8 We discuss related work.

§9 We discuss directions for future work and conclude.

All of our results are machine checked.

2. Localizations

In Figure 1 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` algorithm is good to start with because it is complex enough to require some care to verify while being simple enough that the invariants are straightforward. In §5.2 we will discuss more complex examples that *e.g.* add/change/remove edges and/or vertices.

The code in Figure 1 is written in Clight (Blazy and Leroy 2009), an input language to the CompCert certified compiler (Leroy 2006), which compiles our code exactly as written. The paper-format verification sketch for `mark` in Figure 1 is extracted from a "Floyd" proof in VST (Appel et al. 2014), with only minor cleanup to aid the presentation. Accordingly, there is an unbroken certified chain from our specification of `mark` all the way to the assembly code. In §6 we use HIP/SLEEK (Chin et al. 2010) to verify a Java

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

$$\begin{array}{l}\mathsf{graph}(x,\gamma) \;\Leftrightarrow\; (x = 0 \wedge \mathsf{emp}) \vee \\ \quad \exists m,l,r.\ \gamma(x) = (m,l,r) \wedge x \ \mathsf{mod}\ 16 = 0 \wedge \\ \qquad\qquad x \mapsto m,-,l,r \mathbin{\cup\!\!\!\!\ast} \mathsf{graph}(l,\gamma) \mathbin{\cup\!\!\!\!\ast} \mathsf{graph}(r,\gamma)\end{array} \tag{1}$$

$$\begin{array}{lll}\mathit{mark1}(\gamma,x,\gamma') & \triangleq \forall v.\gamma'(v) = & \begin{cases}(1,l,r) & \text{when } x = v \wedge \\ & \gamma(v) = (0,l,r) \\ \gamma(v) & \text{otherwise}\end{cases} \\ v_1 \overset{\gamma}{\rightsquigarrow}_0 v_2 & \triangleq \exists l,r.\ \gamma(v_1) = (0,l,r) \wedge v_2 \in \{l,r\} & \\ v_1 \overset{\gamma}{\rightsquigarrow}{}_0^{\star} v_2 & \triangleq \text{reflexive, transitive closure of } \overset{\gamma}{\rightsquigarrow}_0 & \\ \mathit{mark}(\gamma,x,\gamma') & \triangleq \forall v.\gamma'(v) = & \begin{cases}(1,l,r) & \text{when } x \overset{\gamma}{\rightsquigarrow}{}_0^{\star} v \wedge \\ & \gamma(v) = (-,l,r) \\ \gamma(v) & \text{otherwise}\end{cases}\end{array}$$

Figure 1. Clight code and proof sketch for bigraph mark.

version of `mark`; the program invariants generated by HIP/SLEEK are slightly different due to HIP/SLEEK's heavier automation.

The specification we certify (lines 5 and 32) is

$$\{\mathsf{graph}(\mathtt{x},\gamma)\}\ \mathtt{mark(x)}\ \{\exists\gamma'.\ \mathsf{graph}(\mathtt{x},\gamma') \wedge \mathit{mark}(\gamma,\mathtt{x},\gamma')\}$$

The specification is for full functional correctness, stated using *mathematical* graphs γ; until §3 consider γ to be a function that maps a vertex $v \in V$ to triples (m,l,r), where m is a "mark" bit (0 or 1) and $\{l,r\} \subseteq V \uplus \{0\}$ are the neighbors of v. The *spatial* graph predicate describes how the mathematical graph γ is implemented in the heap. Until §4 it is enough to know that graph satisfies the fold/unfold relationship in equation (1), located just under the code in Figure 1.

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

Figure 2. Separation logic connectives; $\oplus$ is the join operation on states, usually some kind of disjoint union on heaps

This fold/unfold relationship deserves attention. First, as we explain in §4.1, it is probably a mistake to write (1) as a definition using $\triangleq$ rather than as a biimplication using $\Leftrightarrow$. Second, (1) uses the "overlapping conjunction" $\mathbin{\cup\!\!\!\!\ast}$ of separation logic; informally $P \mathbin{\cup\!\!\!\!\ast} Q$ means that P and Q may overlap in the heap (*e.g.*, nodes in the left subgraph can also be in the right subgraph or even be the root x). The presence of the unspecified sharing indicated by the $\mathbin{\cup\!\!\!\!\ast}$ connective is exactly why graph-manipulating algorithms are so hard to verify (*e.g.*, it is hard to apply the FRAME rule). The standard semantics of the separation logic connectives used in this paper are in Figure 2. Third, (1) illustrates how industrial-strength settings complicate verification. Lines 1–3 define the data type `Node` used by `mark`. The `_Alignas(n)` directives tell CompCert to align fields on n-byte boundaries. As explained in §4.2, this alignment is necessary in C-like memory models to prove fold-unfold (1), which is why (1) includes an alignment restriction x mod $16 = 0$ and an existentially-quantified "blank" second field for the root $x \mapsto m,-,l,r$.

Notice that the postcondition of `mark` is specified *relationally*, *i.e.* $\{\exists\gamma'.\ \mathsf{graph}(\mathtt{x},\gamma') \wedge \mathit{mark}(\gamma,\mathtt{x},\gamma')\}$ instead of *functionally*, *i.e.* $\{\mathsf{graph}(\mathtt{x},\mathit{mark}(\gamma,\mathtt{x}))\}$. 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** the result of marking γ from `x`. For both theoretical and practical reasons a relational approach is better. 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). Our graph `copy` algorithm is specified nondeterministically to avoid specifying how `malloc` allocates fresh blocks of memory. Relations are also preferable to functions because they are more compositional. We take advantage of compositionality by using $\mathit{mark}(\gamma,x,\gamma') \wedge \ldots$ to specify both our "spanning tree" and "graph copy" algorithms in §5.2, which also mark nodes while carrying out their primary task.

Practically, it is painful to define computational functions over graphs in a proof assistant like Coq, and portions of this pain are overkill. For example, Coq requires that all functions terminate, a nontrivial proof obligation over cyclic structures like graphs, but our verification of `mark` is only for partial correctness. Defining relations is much easier because *e.g.* one can use quantifiers and does not have to prove termination. The *mark* and *mark1* relations we use are defined straightforwardly at the bottom of Figure 1.

Turning to the body of the verification (lines 6–31), readers may already have noticed our new notation: blocks of proof sketch bracketed by the symbols $\searrow$ and $\swarrow$, such as lines 10–12. We call a bracketed set of lines like this a "localization block"; localization blocks were inspired by our new `localize` $\searrow$ and `unlocalize` $\swarrow$ tactics in Floyd (§5). The intuitive idea is that we zoom in from a larger "global" context to a smaller "local" one. After verifying some commands locally to arrive at a local postcondition, we zoom back out to the global context. Although we do not do so in Figure 1, localization blocks can safely nest.

In lines 10–12, imagine unfolding the graph predicate in line 9 using equation (1) and then zooming in to the root node x for lines 10–12, before zooming back out in line 13.

To define localization blocks formally we need to first understand the FRAME and RAMIFY rules.

2.1 Frames and ramifications are localizations

The key rule of separation logic is FRAME (Reynolds 2002):

$$\textsc{Frame}\quad \frac{\{P\}\ c\ \{Q\}}{\{P \star F\}\ c\ \{Q \star F\}}\ F \text{ IGNORES } \mathsf{ModVar}(c)$$

The reason FRAME is so important is because it enables local verifications. That is, a verifier can focus on the portions of the heap that are relevant to command c and "frame away" the rest. The side condition "F ignores $\mathsf{ModVar}(c)$" relates to modified program variables and will be discussed in §2.2.

Hobor and Villard observed that FRAME is bit rigid because it forces verifiers to split program assertions into syntactically $\star$-separated parts (Hobor and Villard 2013). This rigidity is particularly unpleasant when verifying programs that manipulate data structures with intrinsic unspecified sharing such as DAGs and graphs. Hobor and Villard proposed the RAMIFY rule to circumvent this rigidity:

$$\textsc{Ramify}\quad \frac{\{L_1\}\ c\ \{L_2\} \qquad G_1 \vdash L_1 \star (L_2 \mathrel{-\!\!\star} G_2)}{\{G_1\}\ c\ \{G_2\}}\ (L_2 \mathrel{-\!\!\star} G_2) \text{ IGNORES } \mathsf{ModVar}(c)$$

That is, we can verify a "global" specification $\{G_1\}\ c\ \{G_2\}$ by combining a "local" specification $\{L_1\}\ c\ \{L_2\}$ with a *ramification entailment* $G_1 \vdash L_1 \star (L_2 \mathrel{-\!\!\star} G_2)$. This entailment uses the "magic wand" operator $\mathrel{-\!\!\star}$ of separation logic[1] to express a notion of "substate update": inside G_1 replace L_1 with L_2 to reach G_2. Essentially the ramification entailment ensures that the change in state specified locally fits properly into the global context. In exchange for proving the ramification entailment, a verifier can use RAMIFY at any time, *i.e.* they need not worry about syntactically matching their assertions with the $\star$ in the FRAME rule. Although the ramification entailments can appear difficult, Hobor and Villard observed that in many practical cases they can be handled easily using a "ramification library".

We are now ready to give a formal meaning to the "localization" pattern employed in Figure 1. When we write:

```
1 // $\{G_1\}$
2 // $\searrow \{L_1\}$
3 $\lightning(i)$    c1; ... ; cn;
4 // $\swarrow \{L_2\}$
5 // $\{G_2\}$
```

we mean apply RAMIFY with $G_1 \vdash L_1 \star (L_2 \mathrel{-\!\!\star} G_2)$. An advantage of this notation is crystal clarity on the predicates used in the ramification entailment. For convenience, the optional $\lightning(i)$ specification can reference an equation or lemma number that solves the ramification entailment. For example, in Figure 1 line 18 references Equation (7) whereas we omit $\lightning$ around line 11 since the heap is unchanged and so the entailment is straightforward. If we wish to save vertical space we can compress the line pairs 1–2 and 4–5 to the single lines $\{G_1\} \searrow \{L_1\}$ and $\{G_2\} \swarrow \{L_2\}$ without sacrificing clarity.

Hobor and Villard pointed out that RAMIFY implies FRAME (modulo the modified program variables issue we fix in §2.2), meaning that our notation can clarify uses of FRAME as well. This is particularly useful in multi-line contexts with nontrivial F, for which the current popular notation to express FRAME involves a liberal use of "…", *e.g.*:

Old notation:	New notation:
`//` $\{P_1 \star F_1 \star F_2 \star F_3\}$	`//` $\{P_1 \star F_1 \star F_2 \star F_3\} \searrow \{P_1\}$
c_1;	c_1;
`//` $\{P_2 \star \ldots\}$	`//` $\{P_2\}$
c_2;	c_2;
`//` $\{P_3 \star \ldots\}$	`//` $\{P_3\}$
c_3;	c_3;
`//` $\{P_4 \star F_1 \star F_2 \star F_3\}$	`//` $\{P_4 \star F_1 \star F_2 \star F_3\} \swarrow \{P_4\}$

2.2 The program variable bugaboo

FRAME's side condition "F ignores $\mathsf{ModVar}(c)$" can be defined in two ways. In the more traditional syntactic style, it means that $\mathsf{FreeVar}(F) \cap \mathsf{ModVar}(c) = \varnothing$. By "syntactic style" we mean that the side condition is written using a function $\mathsf{FreeVar}(F)$ that takes an arbitrary formula and returns the set of free variables within that formula. To define this $\mathsf{FreeVar}(F)$ function we need a fixed inductive **syntax** for formulas. In contrast, in this paper we follow a "semantic style" in which formulas are not given a fixed syntax in advance but can be defined **semantically** on the fly using an appropriate model (Appel et al. 2014). In a semantic style, the side condition on the frame rule is defined as:

$$\begin{aligned} \sigma \overset{S}{\cong} \sigma' &\triangleq \sigma \text{ and } \sigma' \text{ coincide everywhere except } S \\ P \text{ ignores } S &\triangleq \forall \sigma, \sigma'.\ \sigma \overset{S}{\cong} \sigma' \Rightarrow \\ &\qquad (\sigma \models P) \Leftrightarrow (\sigma' \models P) \end{aligned}$$

That is, we consider two program states σ and σ' equivalent up to program variable set S when they agree everywhere except on the values of S (typically, a state σ is a pair of a heap h and program variables ρ). A predicate P ignores S when its truth is independent of all program variables in S.

Now consider using ramification to verify this program:

```
// $\{x = 5 \wedge A\} \searrow \{x = 5 \wedge B\}$
     ...; x = x + 1; ...;
// $\{x = 6 \wedge D\} \swarrow \{x = 6 \wedge C\}$
```

Suppose that the other (elided) lines of the program make localization desirable, even though it is overkill for a single assignment. The key issue is that the program variable x appears in all four positions in the ramification entailment

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

One problem is that $L_2 \mathrel{-\!\!\star} G_2$ does **not** ignore the modified program variable x, preventing us from applying RAMIFY. Intuitively, the side condition on the RAMIFY rule is a bit too strong since it prevents us from mentioning variables in the postconditions that have been modified by code c.

We could try to weaken the side condition in RAMIFY to $\big(\mathsf{FreeVar}(G_2) \cap \mathsf{ModVar}(c)\big) \subseteq \mathsf{FreeVar}(L_2)$, the idea being that information about modified program variables mentioned in the local postcondition L_2 can be carried to the global postcondition G_2. Unfortunately, this idea is unsound because x cannot simultaneously be both 5 and 6, *i.e.* the above entailment is vacuous. A better idea is:

$$\textsc{Ramify-P (Program variables)}\quad \frac{\{L_1\}\ c\ \{L_2\} \qquad G_1 \vdash L_1 \star [\![c]\!](L_2 \mathrel{-\!\!\star} G_2)}{\{G_1\}\ c\ \{G_2\}}$$

The ramification entailment now incorporates a new (universal/boxy) modal operator $[\![c]\!]$. The intuitive meaning of $[\![c]\!]$ is that program variables modified by command c can change value inside its scope. Note that it is vital that L_2 appears as the antecedent

[1] $\mathrel{-\!\!\star}$ is the adjunct of $\star$, *i.e.* $(P \star Q \vdash R) \Leftrightarrow (P \vdash Q \mathrel{-\!\!\star} R)$.

of a (spatial) implication since the change in program variables is universally quantified. This means that if we want to say anything specific about modified program variables in the global postcondition G_2 then we had better say something about them in the local postcondition L_2.

Let us return to our earlier entailment:

$$(\mathtt{x} = 5 \wedge A) \vdash (\mathtt{x} = 5 \wedge B) \ast \\ [\![\mathtt{...;\ x\ =\ x\ +\ 1;\ \ ...;}]\!]((\mathtt{x} = 6 \wedge C) \mathrel{-\!\!\ast} (\mathtt{x} = 6 \wedge D))$$

Since $\mathtt{x}$ is modified, its value can change from the first line, in which $\mathtt{x}$ must be 5, to the second, in which $\mathtt{x}$ must be 6.

Here is the definition of $[\![c]\!]$, writing $\langle c \rangle$ for $\mathsf{ModVar}(c)$:

$$\sigma \models [\![c]\!]P \quad \triangleq \quad \forall \sigma'.\, (\sigma \overset{\langle c \rangle}{\cong} \sigma') \Rightarrow (\sigma' \models P)$$

In other words, $[\![c]\!]$ is exactly the universal modal operator $\Box$ over the relation that considers equivalent all states that differ only on program values modified by c. Since $\overset{\langle c \rangle}{\cong}$ is an equivalence relation, $[\![c]\!]$ forms an S5 modal logic.

Note that RAMIFY-P has no free variable side condition, which is unnecessary because $\forall P.\ [\![c]\!]P$ ignores $\mathsf{ModVar}(c)$. However, in practice this side condition reappears because to actually prove a ramification entailment containing $[\![c]\!]$ one typically applies the following SOLVE RAMIFY-P rule:

$$\textsc{Solve Ramify-P}\quad \frac{G_1 \vdash L_1 \ast F \qquad F \vdash L_2 \mathrel{-\!\!\ast} G_2}{G_1 \vdash L_1 \ast [\![c]\!](L_2 \mathrel{-\!\!\ast} G_2)}\ F \text{ IGNORES } \mathsf{ModVar}(c)$$

We can handle the $[\![c]\!]$ by breaking apart the single entailment into a pair. Using two entailments allows modified program variables to change between the preconditions and postconditions[2]. To connect the pair, we must choose a suitable predicate F that ignores modified variables in c.

With RAMIFY-P and SOLVE RAMIFY-P we can prove the FRAME rule with its canonical side condition as follows:

$$\frac{\dfrac{P \ast F \vdash P \ast F \qquad F \vdash Q \mathrel{-\!\!\ast} (Q \ast F)}{P \ast F \vdash P \ast [\![c]\!](Q \mathrel{-\!\!\ast} (Q \ast F))}\ \begin{array}{l}F \text{ IGNORES} \\ \mathsf{ModVar}(c)\end{array} \qquad \{P\}\ c\ \{Q\}}{\{P \ast F\}\ c\ \{Q \ast F\}}$$

This justifies our point in §2.1 that our new localization notation can also be used for frames.

Choosing F in a concrete setting is a little delicate. For our example, we can just substitute[3] $\mathtt{x}$ for 6 in $L_2 \mathrel{-\!\!\ast} G_2$:

$$F \quad \triangleq \quad (6 = 6 \wedge [\mathtt{x} \mapsto 6]C) \mathrel{-\!\!\ast} (6 = 6 \wedge [\mathtt{x} \mapsto 6]D)$$

The first premise of SOLVE RAMIFY-P is

$$\mathtt{x} = 5 \wedge A \ \vdash\ (\mathtt{x} = 5 \wedge B) \ast \\ ((6 = 6 \wedge [\mathtt{x} \mapsto 6]C) \mathrel{-\!\!\ast} (6 = 6 \wedge [\mathtt{x} \mapsto 6]D))$$

This entailment is the key proof that our localization was sound. Generally speaking this entailment is solved by using a ramification library (§4.3); as previously explained we sometimes use $\lightning(n)$ to explicitly reference a library lemma.

Meanwhile, the second premise looks like this:

$$(6 = 6 \wedge [\mathtt{x} \mapsto 6]C) \mathrel{-\!\!\ast} (6 = 6 \wedge [\mathtt{x} \mapsto 6]D) \ \vdash \\ (\mathtt{x} = 6 \wedge C) \mathrel{-\!\!\ast} (\mathtt{x} = 6 \wedge D) \qquad (2)$$

Although it may not be readily apparent, this is in fact a tautology using $(P \ast Q \vdash R) \Leftrightarrow (P \vdash Q \mathrel{-\!\!\ast} R)$.

[2] Entailment procedures for separation logic may prefer to use $F \ast L_2 \vdash G_2$ as the second premise of SOLVE RAMIFY-P because it is free from $\mathrel{-\!\!\ast}$.

[3] In a semantic setting, substitution is defined with a modal operator rather than textual replacement, but the net effect is the same.

This strategy is sufficient to handle all of the localization blocks in Figure 1. For example, in lines 16–20, choose $F \triangleq$

$$(\mathtt{x} \mapsto 1, -, l, r \wedge \gamma(\mathtt{x}) = (0, l, r) \wedge \exists \gamma'.\ \mathit{mark1}(\gamma, \mathtt{x}, \gamma')) \\ \mathrel{-\!\!\ast} (\exists \gamma'.\ \mathsf{graph}(\mathtt{x}, \gamma') \wedge \gamma(\mathtt{x}) = (0, l, r) \wedge \mathit{mark1}(\gamma, \mathtt{x}, \gamma'))$$

Note the use of the metavariables l and r rather than $\mathtt{l}$ and $\mathtt{r}$ in F, added to the metacontext in lines 8–9 using Floyd's EXISTENTIAL EXTRACTION rule (Floyd 1967):

$$\textsc{Existential extraction}\quad \frac{\forall x.\ (\{P\}\ c\ \{Q\})}{\{\exists x. P\}\ c\ \{\exists x.\ Q\}}$$

Pen and paper Hoare proofs are often a little casual with existentials, *e.g.* omitting line 8; we wrote it because we wanted to be clear that the metavariables l and r were properly "in scope" over the localization blocks.

2.3 The existential ogre

What happens when we **cannot** calculate a substitution using globally-scoped metavariables? Consider the following:

```
1 // $\{A\} \searrow \{B\}$
2       ...; x = malloc(sizeof(int));
3       if (x == 0) then y = 0 else y = 1; ...;
4 // $\swarrow \{((\mathtt{x} \mapsto - \wedge \mathtt{y} = 1) \vee (\mathtt{x} = 0 \wedge \mathtt{y} = 0)) \ast C\}$
5 // $\{(\mathtt{y} = 1 \wedge D_1) \vee (\mathtt{y} = 0 \wedge D_2)\}$
```

Within a localization block we call the nondeterministically specified function `malloc` and use the program variable `y` as a flag to keep track of whether the allocation succeeded. Call the postconditions in lines 4 and 5 just above L_2 and G_2 respectively.

Now the choice of F is not very straightforward because we do not know the values to substitute for `x` or `y`:

$$[\mathtt{x} \mapsto ?][\mathtt{y} \mapsto ?](L_2 \mathrel{-\!\!\ast} G_2) \qquad (3)$$

We can avoid this roadblock as follows. First, rewrite the postconditions in lines 4 and 5 just above to introduce fresh existentially-quantified variables x and y and bind them to `x` and `y`:

```
4 //     $\{L_2\}$
5 // $\swarrow \{\exists x, y.\ x = \mathtt{x} \wedge y = \mathtt{y} \wedge [\mathtt{x} \mapsto x][\mathtt{y} \mapsto y]L_2\}$
6 // $\{\exists x, y.\ x = \mathtt{x} \wedge y = \mathtt{y} \wedge [\mathtt{x} \mapsto x][\mathtt{y} \mapsto y]G_2\}$
7 // $\{G_2\}$
```

Call these equivalent postconditions L_2' (line 5) and G_2' (line 6).

Next apply RAMIFY-P and SOLVE RAMIFY-P with $F \triangleq$

$$\forall x, y.\ [\mathtt{x} \mapsto x][\mathtt{y} \mapsto y](L_2 \mathrel{-\!\!\ast} G_2)$$

In other words, replace the "?" from (3) with universally-quantified metavariables x and y scoped over the entire $\mathrel{-\!\!\ast}$.

Now consider the first premise of SOLVE RAMIFY-P:

$$G_1 \vdash L_1 \ast F \ \Big|\ A \vdash B \ast \forall x, y.\ [\mathtt{x} \mapsto x][\mathtt{y} \mapsto y](L_2 \mathrel{-\!\!\ast} G_2)$$

This is essentially the same ramification entailment we had before, and so the general strategy is to apply the ramification library §4.3. The second premise is more interesting:

$$\left.\begin{array}{l} F \vdash \\ (L_2' \mathrel{-\!\!\ast} \\ G_2') \end{array}\right| \begin{array}{c} (\forall x, y.\ [\mathtt{x} \mapsto x][\mathtt{y} \mapsto y](L_2 \mathrel{-\!\!\ast} G_2)) \vdash \\ (\exists x, y.\, x = \mathtt{x} \wedge y = \mathtt{y} \wedge [\mathtt{x} \mapsto x][\mathtt{y} \mapsto y]L_2) \mathrel{-\!\!\ast} \\ (\exists x, y.\, x = \mathtt{x} \wedge y = \mathtt{y} \wedge [\mathtt{x} \mapsto x][\mathtt{y} \mapsto y]G_2) \end{array}$$

Like equation (2), this turns out to also be a tautology, albeit a more complicated one. Since L_2 and G_2 are equivalent to L_2' and G_2', we can therefore verify the specification all the way from A to G_2 despite the presence of the existentially-quantified modifications to the program variables `x` and `y`.

We package all of this reasoning into the following rule:

$$\textsc{Ramify-PQ (Program variables and Quantifiers)} \\ \frac{\{L\}\ c\ \{\exists x.\ L_2\} \qquad G_1 \vdash L_1 \ast [\![c]\!](\forall x.\ (L_2 \mathrel{-\!\!\ast} G_2))}{\{G_1\}\ c\ \{\exists x.\ G_2\}}$$

Proof of RAMIFY-P from FRAME and CONSEQUENCE:

$$\dfrac{G_1 \vdash L_1 * \llbracket c \rrbracket (L_2 -\!\!* G_2) \qquad \dfrac{\{L_1\}\ c\ \{L_2\}}{\{L_1 * \llbracket c \rrbracket (L_2 -\!\!* G_2)\}\ c\ \{L_2 * \llbracket c \rrbracket (L_2 -\!\!* G_2)\}}\,(1) \qquad \dfrac{\dfrac{\overset{\langle c \rangle}{\cong} \text{ is reflexive}}{\llbracket c \rrbracket (L_2 -\!\!* G_2) \vdash L_2 -\!\!* G_2}\,(2)}{L_2 * \llbracket c \rrbracket (L_2 -\!\!* G_2) \vdash G_2}\,(3)}{\{G_1\}\ c\ \{G_2\}}$$

(1) $\forall P.\ \llbracket c \rrbracket P$ ignores $\mathsf{FreeVar}(c)$ (2) axiom T of modal logic (3) $(P * Q \vdash R) \Leftrightarrow (P \vdash Q -\!\!* R)$

Proof of RAMIFY-PQ from RAMIFY-P:

$$\dfrac{\{L_1\}\ c\ \{\exists x.\ L_2\} \qquad \dfrac{G_1 \vdash L_1 * \llbracket c \rrbracket \big(\forall x.\ (L_2 -\!\!* G_2)\big) \qquad \dfrac{\dfrac{\dfrac{\vdots}{\forall x.\ (L_2 -\!\!* G_2) \vdash (\exists x.\ L_2) -\!\!* (\exists x.\ G_2)}\,(1)}{\llbracket c \rrbracket \big(\forall x.\ (L_2 -\!\!* G_2)\big) \vdash \llbracket c \rrbracket \big((\exists x.\ L_2) -\!\!* (\exists x.\ G_2)\big)}\,(2)}{L_1 * \llbracket c \rrbracket \big(\forall x.\ (L_2 -\!\!* G_2)\big) \vdash L_1 * \llbracket c \rrbracket \big((\exists x.\ L_2) -\!\!* (\exists x.\ G_2)\big)}}{G_1 \vdash L_1 * \llbracket c \rrbracket \big((\exists x.\ L_2) -\!\!* (\exists x.\ G_2)\big)}}{\{G_1\}\ c\ \{\exists x.\ G_2\}}$$

(1) tautology using $(P * Q \vdash R) \Leftrightarrow (P \vdash Q -\!\!* R)$ (2) reduction using modal axioms K and N

Figure 3. Proofs of RAMIFY-P and RAMIFY-PQ

Essentially RAMIFY-PQ allows us to shift existential variables from the local context to the global one in a smooth way, especially in conjunction with the following rule:

SOLVE RAMIFY-PQ

$$\dfrac{G_1 \vdash L_1 * F \qquad F \vdash \forall x.\ (L_2 -\!\!* G_2)}{G_1 \vdash L_1 * \llbracket c \rrbracket \big(\forall x.\ (L_2 -\!\!* G_2)\big)} \quad F \text{ IGNORES } \mathsf{ModVar}(c)$$

Since we use a relational style to verify graph algorithms (*e.g.* in Figure 1), existentials appear frequently and a smooth treatment is very helpful in practice. To make this point a little more clearly we were more explicit about existentials in *e.g.* lines 22–26 than is typical in pen-and-paper proofs. However, fortified by the RAMIFY-PQ rule, we could very reasonably have *e.g.* written line 25 as

```
25 // ↙ {graph(1, γ″) ∧ mark(γ′, 1, γ″)}
```

and omitted line 26 entirely.

Although our technique to handle modified program variables is rather intricate, it can be done mechanically/automatically (§5). Our `localize` and `unlocalize` tactics use RAMIFY-PQ since it is the most general rule.

2.4 Soundness of our rules

In Figure 3 we sketch the soundness proofs for RAMIFY-P and RAMIFY-PQ. RAMIFY-P requires only FRAME and CONSEQUENCE to prove, along with some basic properties of $\llbracket c \rrbracket$. RAMIFY-PQ is built on top of RAMIFY-P with some complicated logical maneuvers. Systems of separation logic that do not wish to add $\llbracket c \rrbracket$ to their logical formulae might consider adding a rule that packages the RAMIFY-PQ and SOLVE RAMIFY-PQ rules together.

3. A framework for graph theory

To enable the verification of full functional correctness of graph algorithms we need a way to reason about mathematical graphs. To allow such verifications to be mechanized without undue pain we must take care to develop a modular and general-purpose framework for such mathematical graphs.

3.1 Structure of the mathematical graph framework

Figure 4 gives the overall architecture of how our graphs are constructed. The most basic kind of graph is PreGraph, out of which we build LabeledGraphs, and which in turn are used to build GeneralGraphs. Each kind of graph has some associated lemmas, and each kind inherits the lemmas of the previous kind. The dashed box represents a "plugin" system for attaching arbitrary properties to LabeledGraphs and will be discussed more later.

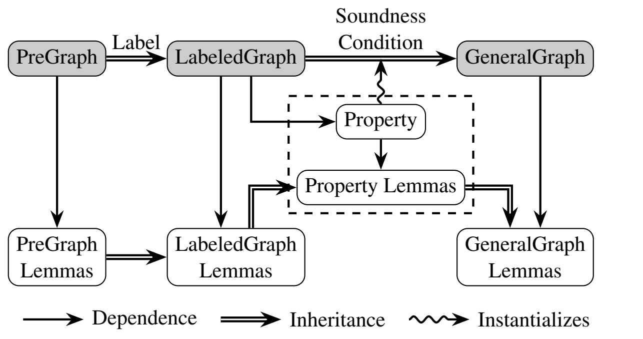


Figure 4. Structure of the Mathematical Graph Library

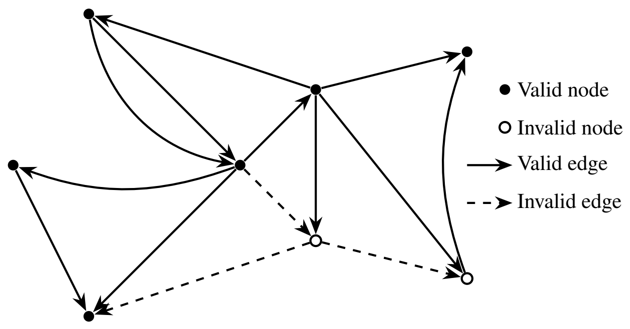


Figure 5. A PreGraph with invalid nodes and edges.

Pregraphs. A PreGraph is a hextuple $(V, E, \phi_V, \phi_E, s, t)$, where V and E are the underlying carrier set of vertices and edges. Not every $v \in V$ or $e \in E$ is actually "in" the graph, so we provide the predicates ϕ_V and ϕ_E to classify vertices and edges as *valid* (in) or not (out). Finally, s and $e : E \to V$ are functions that map an edges to their source and destination respectively; this model means that PreGraphs are directed rather than undirected. By design, there are no requirements for *e.g.* how the validities of edges and vertices relate. As shown in Figure 5, a PreGraph can contain invalid nodes and edges in an arbitrary configuration.

The advantage of designing a graph type that can reason about missing vertices and edges is because some of our later definitions need such flexibility. Consider the difference of two graphs, $\gamma_1 - \gamma_2$. Even if both of these graphs are "well-formed" to begin with, in the sense that valid nodes have only valid edges and vice versa, their difference may not since there may be dangling edges pointing to the now-removed vertices of γ_2.

Many basic graph concepts such as *path*, *reachability*, and *subgraph* are defined on PreGraphs. Informally a path is a list of nodes connected by edges. Formally it is more convenient to define a path as an ordered pair (n, l) where n is a node and l is a list of edges. A valid path requires n to be the source of the first edge of l (if one exists) and moreover requires l to be "well chained". That is, the destination of one edge in l must be the source of the next edge. The list l can be null to represent an empty path starting and ending at the node n. We prefer this encoding as opposed to some others (*e.g.* a list of edges) because the definitions of important concepts like reachability are cleaner.

The most important definition on PreGraph is the concept of reachability. We use the notation $\gamma \models L^{n_1}_{n_2}(P)$ to mean that we can reach n_2 from n_1 via the path L in PreGraph γ, and moreover that every node in L satisfies predicate P. This notation, along with some derived ones such as

$$\gamma \models n_1 \xrightarrow{P} n_2 \triangleq \exists L, \gamma \models L^{n_1}_{n_2}(P),$$
$$\gamma \models n_1 \leadsto n_2 \triangleq \exists L, \gamma \models L^{n_1}_{n_2}(True)$$

form the bedrock of nearly every nontrivial predicate about and relation between graphs. We write $\mathsf{reachable}(\gamma, v)$ to mean the set of vertices reachable in γ from a given vertex v.

In §4 we will tie mathematical graphs γ to a spatial graph predicate $\mathsf{graph}(x, \gamma)$. As we will see, graph "owns" only the spatial portion of γ that is reachable from x even though γ may contain other nodes. Accordingly, when we reason about $\mathsf{graph}(x, \gamma)$, it is natural to want to describe the reachable portion of γ. In fact we generalize this idea into two concepts: the subgraph of γ satisfying an arbitrary predicate P, written $\gamma \downarrow P$, and the *relaxed subgraph*, written $\gamma \uparrow P$, which contains the all of the subgraph plus some additional edges. In particular, $\gamma \downarrow P$ contains exactly the vertices satisfying P and only the edges whose source and destination both satisfy P. The relaxed subgraph $\gamma \uparrow P$ adds the additional edges whose **source** satisfies P, even though their destination may not. We can use these definitions to, for example, extract the subgraph or relaxed subgraph reachable from a vertex v by writing *e.g.*

$$\gamma \downarrow (\lambda v'. \gamma \models v \leadsto v')$$

LabeledGraph. PreGraph and its derived properties (reachability, subgraph, etc) are inadequate for real program verification, even though many basic lemmas can already be proved about them. However, when reasoning about the concrete graphs manipulated by various algorithms, we usually need to add a notion of *labels* on vertices and/or edges, such as the "mark bit" used in Figure 1.

GeneralGraph. Much more interesting is the concept of GeneralGraph, which augments a LabeledGraph by adding a user-specified soundness condition. In Figure 4 this soundness condition is highlighted by a dashed border. These "plugins" can specify many different kinds of properties. Each property, in turn, can be used to prove many property-specific lemmas, all of which then apply to the instantiating GeneralGraph.

3.2 Graph plugins

Many theorems about graphs require certain properties, such as:

- A graph may be finite (FiniteGraph), meaning that both the set of valid vertices and the set of valid edges are finite.
- A less restrictive property is *locally finite* (LocalFiniteGraph), in which each vertex has a finite number of neighbors. Locally finite graphs are useful, for example, when we wish to reason about algorithms that process vertices by cycling through neighbors, such as breadth-first search.

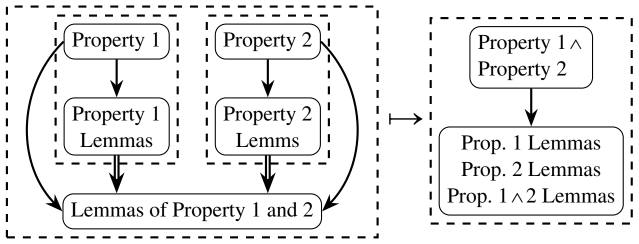


Figure 6. Combining plugins

- More subtly, consider that many real data structures use special null values to represent unused edges. The MathGraph property introduces this concept—*i.e.* some invalid nodes that are none-the-less allowed to appear as destinations for valid edges.
- Many of our verified algorithms have only two outgoing edges per node. The BiGraph property lets us reason about this common special case in a convenient manner.

We have a number of other properties in our codebase, but these four are the most important basic ones.

We use Coq's typeclass system to manage our plugins in a smooth manner. Essentially the typeclass system enables the diagram in Figure 6. The idea is that if we have two properties, each of which come with some already-proved lemmas, we can combine these plugins to prove the emergent lemmas that result from the combination, and then treat the new combination as a new plugin. Since the system is compositional, we can easily mix many different properties together. For example, we compose BiGraph, MathGraph, and FiniteGraph together into a new plugin we call BiMaFin. BiMaFin is the actual soundness condition used to verify the program in Figure 1.

One interesting example of this process is the following lemma:

LEMMA 1 (Computable reachability).

$$\begin{aligned}&\forall \gamma, x.\ \mathsf{MathGraph}\ \gamma \Rightarrow \mathsf{LocalFiniteGraph}\ \gamma \Rightarrow\\ &\quad \mathsf{reachable}(\gamma, x)\ \textit{is finite} \Rightarrow\\ &\quad \textit{we can } \textbf{\textit{compute}} \textit{ a set } S \textit{ s.t.}\\ &\qquad \forall v.\ v \in S \Leftrightarrow \gamma \models x \leadsto v.\end{aligned}$$

The proof of this lemma is a bit subtle. To compute the reachable set we need to design a decision procedure that explores the (potentially cyclic) graph. Accordingly we implement a flavor of breadth-first search, also keeping track of the nodes that we have explored previously. Since nodes are explored in BFS order while reachability is in some sense defined in DFS order, when we reach a node n we must take some care to reconstruct a path from the origin to n to satisfy the reachability property. The definition is also painful because of Coq's insistence that all computations terminate; we utilized the techniques outlined by Chlipala to pacify Coq's termination checker (Chlipala 2013).

3.3 Reasoning about relations between graphs

We apply our framework to define other structures and relations between these structures (including graph), and then use them to prove certain pure facts proposed by real verifications.

For example, we define DAG (directed acyclic graph) as a PreGraph with an additional property: forall any x and y, if x is reachable from y, then $x = y$ or y is not reachable from x. Similarly, we define tree by saying that for any reachable node n there is a unique path from the root to n.

We already defined the relation $mark(\gamma, \mathrm{x}, \gamma')$, used in the graph marking algorithm, in Figure 1. Similarly, we define *span* for the spanning tree program and *copy* for the graph copy program. These relations all capture how the graph has changed from before to after the program execution. As previously mentioned, we reuse *mark* and its related lemmas to prove facts about spanning tree and graph copy because the latter two programs mark nodes as they work. Accordingly, we can reuse the following fact:

$$\forall \gamma, x, n.\ \gamma(x) = (0, v_1, v_2, \ldots, v_n) \Rightarrow \mathit{mark1}(\gamma, x, \gamma_1) \Rightarrow \mathit{mark}(\gamma_1, v_1, \gamma_2) \Rightarrow \mathit{mark}(\gamma_2, v_2, \gamma_3) \Rightarrow \cdots \Rightarrow \mathit{mark}(\gamma_n, v_n, \gamma_{n+1}) \Rightarrow \mathit{mark}(\gamma, x, \gamma_{n+1}).$$

This is a general theorem for any $\mathsf{LocalFiniteGraph}$, not just $\mathsf{BiGraph}$. In our framework, we aspire to prove theorems as generally as possible.

4. Defining and reasoning about spatial graphs

To prove the functional correctness of real graph-manipulating algorithms implemented in a real language, we need to connect the heap representation of graphs, the memory model of the programming language, and the mathematical properties of graphs from §3. The first of these turns out to be surprisingly subtle as we shall see in §4.1 and §4.2. The main challenge for the others 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 §4.3.

4.1 Traditional fixpoints fail to define good graph predicates

Recursive predicates are ubiquitous in separation logic—so much so that when a person 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 what is going on is instead of defining P directly, one defines a functional $F_P \triangleq \lambda P.\ \ldots P \ldots$ and then defines P itself as $P \triangleq \mu_{\mathsf{T}}\, F_P$. Assuming (as one typically does without comment) that F_P is *covariant*, i.e. $(P \vdash Q) \Rightarrow (F\, P \vdash F\, Q)$, one then enjoys the fixpoint equation $P \Leftrightarrow \ \ldots P \ldots$, formally justifying typically written pseudodefinition ("$\triangleq$").

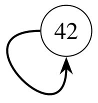
Suppose we define a graph predicate graph_T this way, *e.g.* along the lines of the fold/unfold definition we used in Figure 1, *i.e.*

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

We have removed the alignment-related portions of equation (1) to focus on a more serious issue, even though as we will explain in §4.2 alignment concerns are also necessary for the fold/unfold relationship to hold in C-like memory models. The functional needed to define graph_T is covariant, so we can apply Knaster-Tarski soundly. However, the resulting predicate can be hard to use.

Consider the following partial memory m for a toy machine:

address	value
102	0
101	100
100	42



Clearly $m \models 100 \mapsto 42, 100, 0$. But it seems also clear that this memory represents a one-cell cyclic graph as illustrated in the accompanying diagram, *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 hinted at in Figure 7, this seems rather difficult to do so since applying the natural proof techniques have only strengthened the goal. In fact we do not know if this entailment is provable or not, 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.

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. Note that 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 §4.2 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 very easy to prove.

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 the "approximation modality" $\triangleright$ (Appel et al. 2007), *i.e.* our graph predicate would look like

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

As we will see in §6, the forward style of reasoning employed by HIP/SLEEK is greatly aided when predicates are *precise*, *i.e.*

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

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 mentioned previously in the literature.

4.2 Defining a good graph predicate

We choose an alternative path. Rather than trying to define graph directly as a recursive fixpoint, we will give it a flat structure and then **prove** that it satisfies fold/unfold. Our path starts with the iterated separating conjunction or "big star", defined as follows:

$$\bigstar_{\{l_1, l_2, \ldots, l_n\}} P \triangleq P(l_1) * P(l_2) * \ldots * P(l_n).$$

Notice that formally $\bigstar$ is defined over a list rather than a set, and is parameterized by a predicate P. It is natural to extend it to a set S using an existentially-quantified duplicate-free list L as follows:

$$\bigstar_S P \triangleq \exists L.\ (\mathsf{NoDup}\ L) \wedge (\forall x.\ x \text{ in } L \Leftrightarrow x \in S) \wedge \bigstar_L P$$

We use the same $\bigstar$ notation since the concepts are similar, but the existential can add a little pain since it means that we need to prove that all choices of list L yield equivalent predicates.

We are now ready to give a good graph predicate:

$$\mathsf{graph}(x, \gamma) \triangleq \bigstar_{v \in \mathit{reach}(\gamma, x)} v \mapsto \gamma(v) \qquad (4)$$

Here γ is a GeneralGraph from §3.1 and "$x \mapsto \gamma(x)$" is a predicate that describes how single nodes fit in memory; in Figure 1 it was

$$\exists m, l, r.\ \gamma(x) = (m, l, r) \wedge x \mapsto m, -, l, r \wedge x \ \mathsf{mod}\ 16 = 0$$

In general γ 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, assuming our graph and GeneralGraph give us the necessary properties. To state a general fold/unfold lemma we need the iterated overlapping conjunction:

$$\biguplus_{l_1, \ldots, l_n} P \triangleq P(l_1) \uplus \ldots \uplus P(l_n)$$

Now we can state the general fold/unfold as follows:

$$\mathsf{graph}(x, \gamma) \Leftrightarrow x \mapsto \gamma(x) \uplus \Big(\biguplus_{n \in \mathsf{neighbors}(\gamma, x)} \mathsf{graph}(\gamma, n) \Big) \qquad (5)$$

$$\dfrac{\dfrac{100 \mapsto 42, 100, 0 \;\vdash\; 100 \mapsto 42, 100, 0 \mathbin{\cup\!\!\!\!*} \mathsf{graph}_T(100, \hat{\gamma})}{100 \mapsto 42, 100, 0 \;\vdash\; \hat{\gamma}(100) = (42, 100, 0) \;\wedge\; 100 \mapsto 42, 100, 0 \mathbin{\cup\!\!\!\!*} \mathsf{graph}_T(100, \hat{\gamma}) \mathbin{\cup\!\!\!\!*} \mathsf{graph}_T(0, \hat{\gamma})}\;(2)}{100 \mapsto 42, 100, 0 \;\vdash\; \mathsf{graph}_T(100, \hat{\gamma})}\;(1)$$

(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

Figure 7. An honest academic tries to prove a "simple" entailment

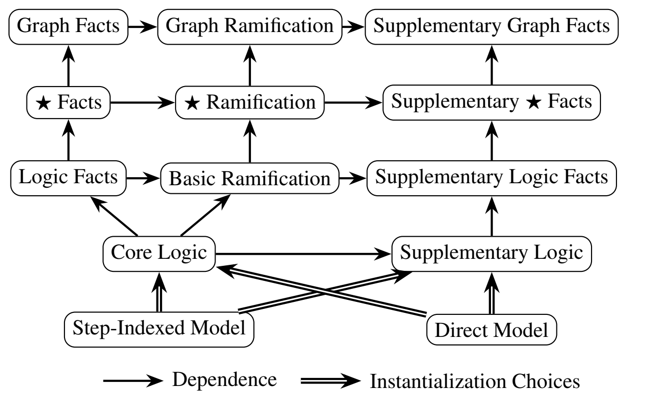


Figure 8. Infrastructure of ramification library

In other words, we get a full equivalence between a graph and its "unfolded" structure, regardless of how many neighbors x has.

Both directions of this lemma are a little subtle. In the $\Rightarrow$ direction, the key difficulty is that we need to take the existentially-quantified list of vertices on the left L and divide it into (not necessarily disjoint) sublists $L_1, \ldots, L_n$ such that L_i is exactly those vertices reachable from neighbor i. To construct these lists we need to use the computable reachability lemma 1 to explicitly test, for each neighbor i and v in L, whether i can reach v; accordingly we require that the GeneralGraph be appropriately finite.

In the $\Leftarrow$ direction, 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 $\bigstar$ on the left hand side cannot separate them. To avoid skewing we require $x \mapsto \gamma(x)$ be *alignable*. A predicate P is alignable when

$$\forall x, y.\ \Big(P(x) \mathbin{\cup\!\!\!\!*} P(y) \vdash \big(P(x) \wedge x = y\big) \vee \big(P(x) * P(y)\big)\Big)$$

In other words, either they are completely on top of one another or they do not interfere at all. In a Java-like memory model such as in HIP/SLEEK 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 in Figure 1).

Some of our VST proofs do not use fold/unfold, instead preferring to use the lemmas in §4.3 directly. On the other hand, for HIP/SLEEK fold/unfold is vital, and knowing that the recursive relationship holds produces a pleasant feeling. We also prove a general fold/unfold lemma for DAGs in which we get a $*$ between the root and its $\boxed{\cup\!\!\!\!*}$-joined neighbors rather than the $\mathbin{\cup\!\!\!\!*}$ present in (5).

4.3 Ramification Libraries

We provide of our spatial development in Figure 8. Starting from the bottom, notice that there are two underlying heap models: the Step-Indexed Model, which is the main heap model used in VST, and a much simpler Direct Model, which is used by HIP/SLEEK among others. The Step-Indexed model is much fancier, but none of our development depends on its bells and whistles.

To isolate our development from these unnecessary complications, and to ensure that HIP/SLEEK can reuse our spatial reasoning, we use two interfaces: Core Logic and Supplementary Logic. Both models can instantiate both interfaces, but generally speaking our VST proofs only need the Core properties to prove our examples, whereas HIP/SLEEK uses both Core and Supplemental. Each interface defines some operators of separation logic and provides some axioms about how they work. For example, $*$ and $-\!\!*$ are in Core Logic, along with the axiom $(P \vdash Q -\!\!* R) \Leftrightarrow (P {*} Q \vdash R)$. On the other hand, the $\mathbin{\cup\!\!\!\!*}$ and $-\!\!\circledast$ operators are in Supplementary Logic, along with rules like $P \vdash P \mathbin{\cup\!\!\!\!*} P$.

Above the Logic layer we have three towers, each three levels high. The leftmost tower is about basic lemmas about Logic, $\bigstar$, and graph. For example, in the $\bigstar$ Facts box we prove lemmas such as the following:

$$\dfrac{A \cap B = \varnothing}{\bigstar_{x \in A} P(x) * \bigstar_{x \in B} P(x) \Leftrightarrow \bigstar_{x \in A \cup B} P(x)}$$

The middle tower is more interesting in that it is entirely focused on ramification entailments. A robust library of ramification entailments is essential to making ramification work smoothly in practice. The lowest level contains lemmas such as the following:

$$\textsc{Ramify-Q-Split}\quad \dfrac{G_1 \vdash L_1 * \forall x.\ (L_2 -\!\!* G_2) \qquad G_1' \vdash L_1' * \forall x.\ (L_2' -\!\!* G_2')}{G_1 * G_1' \vdash (L_1 * L_1') * \forall x.\ \big((L_2 * L_2') -\!\!* (G_2 * G_2')\big)}$$

We use this lemma to break large ramification entailments into more manageable pieces in a compositional way.

The middle level contains ramification lemmas about $\bigstar$, such as the following:

$$\dfrac{A \cap B = \varnothing, \quad A' \cap B = \varnothing}{\bigstar_{x \in A \cup B} P(x) \vdash \bigstar_{x \in A} P(x) * \Big(\bigstar_{x \in A'} P(x) -\!\!* \bigstar_{x \in A' \cup B} P(x)\Big)} \qquad (6)$$

The top level is focused on graph ramifications, such as the following "update one node" lemma:

$$\dfrac{\begin{array}{c}\forall x_0 \neq x.\ \gamma(x_0) = \gamma'(x_0) \\ \mathsf{neighbors}(\gamma, x) = \mathsf{neighbors}(\gamma', x)\end{array}}{\mathsf{graph}(x, \gamma) \vdash x \mapsto \gamma(x) * \big(x \mapsto \gamma'(x) -\!\!* \mathsf{graph}(x, \gamma')\big)} \qquad (7)$$

This lemma was used on line 18 in Figure 1.

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 $\bigstar$ to be reused in other structures smoothly.

Also, all of 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

1	$\{ \ P_1 \ \}$	$\{ \ P_1 \ \}$	$\{ \ P_1 \ \}$	$\{ \ P_1 \ \}$
2	`c1`	`c1`	`c1`	`c1`
3	$\{ \ P_2 \ \}$	$\{ \ P_2 \ \}$	$\{ \ P_2 \ \}$	$\{ \ P_2 \ \}$
4	$\searrow \{ \ P_3 \ \}$	$\{ \ ?F \ast P_3 \ \}$	$\searrow \{ \ P_3 \ \}$	$\{ \ ?F \ast P_3 \ \}$
5	`c2;`	`c2;`	`c2;`	`c2;`
6	$\{ \ P_4 \ \}$	$\{ \ ?F \ast P_4 \ \}$	$\{ \ P_4 \ \}$	$\{ \ ?F \ast P_4 \ \}$
7			`c3;`	`c3;`
8	$\ldots$	$\ldots$	$\swarrow \{ \ P_5 \ \}$	$\{ \ ?F \ast P_5 \ \}$
9			$\{ \ P_6 \ \}$	$\{ \ P_6 \ \}$
10				
11			$\ldots$	$\ldots$

Figure 9. Front and back ends of `localize` and `unlocalize`

the following theorem from the library:

$$\frac{n \in \mathsf{neighbors}(\gamma, x)}{\begin{array}{l}\mathsf{graph}(x,\gamma) \vdash \\ \quad \mathsf{graph}(n,\gamma) \ast (\forall \gamma'.\ \mathit{mark}(\gamma,n,\gamma') \wedge \mathsf{graph}(n,\gamma') \mathrel{-\!\!\ast} \\ \qquad\qquad\qquad \mathit{mark}(\gamma,n,\gamma') \wedge \mathsf{graph}(x,\gamma'))\end{array}} \tag{8}$$

The Supplementary tower contains properties not used by most of the VST examples. This includes the fold/unfold relationship from §4.2, facts about precision, and so forth. As we will see in §6, some of these properties are needed by HIP/SLEEK. Other supplementary lemmas are mostly included for esthetic effect.

5. Ramification in VST

The Verified Software Toolchain is a series of machine-checked modules written in Coq whose focus is reasoning about C programs (Appel et al. 2014), especially those programs that can be compiled with the CompCert compiler (Leroy 2006). One of VST's modules, Floyd, is a separation-logic based engine to help users verify concrete programs. The modules interlock so there are no "gaps" in the end-to-end certified results; accordingly all of the rules employed by Floyd have been proved sound with respect to the underlying semantics used by CompCert. Floyd is written in a combination of Ltac and Gallina and is designed to help users verify the full functional correctness of their programs. Although Floyd devotes considerable effort to make this task simpler, it prefers expressibility and completeness to more automated tools like HIP/SLEEK.

Floyd presents users with a pleasant "decorated program" visualization for Hoare proofs, in which users work from the top of the program to the bottom even though the formal proof is maintained as applications of inference rules. For example, suppose the proof goal is $\{P_1\}\, c_1\, ;\, c_2\, \{P_5\}$ and VST's user tells Floyd to apply a Hoare rule for c_1, *e.g.* $\{P_1\}\, c_1\, \{P_2\}$. Floyd will then automatically apply 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") and the inference tree is also partially done (*i.e.* with holes that are represented by the remaining proof goals in Coq).

5.1 The `localize` and `unlocalize` tactics

We wish to preserve this "decorated program" view while extending Floyd to support ramification. 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 9 has four 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 the assertion P_2 (line 3) and decided to use the `localize` tactic to zoom into P_3 (line 4). They then applied 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 how 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 9 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 $[\![c_2\,;c_3]\!](P_5 \mathrel{-\!\!\ast} P_6)$. In fact, as discussed in §2.3, `unlocalize` replaces P_5 and P_6 with equivalent P_5' and P_6' that use existentials to pack up modified program variables, and then instantiates $?F$ with a version of $[\![c_2\,;c_3]\!](P_5 \mathrel{-\!\!\ast} P_6)$ in which those same modified program variables have been replaced with universally-quantified metavariables. Reformulating assertions to isolate program variables is aided by Floyd's use of a canonical form for assertions that explicitly separates assertions containing program variables from spatial assertions.

As indicated in §2.3, this leaves two proof goals. The first, $P_2 \vdash ?F \ast P_3$, is simplified to remove the $[\![c_2\,;c_3]\!]$ and referred to the user as a new obligation; most often the user solves it using a ramification library (§4.3). The second, $?F \vdash P_5' \mathrel{-\!\!\ast} P_6'$, is solved automatically by `unlocalize`.

5.2 Additional examples in VST

In Figure 10 we show a simplified proof script for the spanning tree algorithm. We have also verified `mark` for DAGs and `copy` for graphs. For the sake of space, we do not put these decorated program scripts here.

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.

6. Ramification in HIP/SLEEK

HIP/SLEEK is a toolset for verifying programs using separation logic (Chin et al. 2010). As compared to VST, H/S has a heavier focus on automation: for example, users need only specify loop invariants and the pre/postconditions of methods, rather than describing each program point. H/S has two interlocking components. HIP applies Hoare rules using forward reasoning to verify programs, *i.e.* each entailment is of the form $P \vdash Q \ast R$, where the heap from the antecedent P is matched with the consequent Q and a frame/residue R. To check these separation logic entailments and calculate R, HIP calls SLEEK. SLEEK handles spatial operators such as $\ast$ and $\mapsto$ before handing any remaining pure entailments to a variety of external solvers such as Z3. One of H/S's distinguishing features is support for user-defined recursive predicates. HIP/SLEEK has been used to verify programs manipulating data structures like lists, arrays, and trees.

6.1 Verifying `mark` in HIP/SLEEK

Figure 11 gives most of the HIP/SLEEK file used to verify `mark` (we have elided about 15 lines to save space). We specify the pre- and postcondition of `mark` in line 33 and 34) respectively; notice that this is the same specification we verified in VST in Figure 1. It is quite unusual for H/S to verify the full functional correctness

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

// We use $R$ to represent $\mathsf{reachable}(\gamma, \mathtt{x})$

void spanning(struct Node * x) {// $\{\mathsf{graph}(\mathtt{x},\gamma) \wedge \gamma(\mathtt{x}).1 = 0\}$
    struct Node * l, * r; int root_mark;
// $\{\mathsf{graph}(\mathtt{x},\gamma) \wedge \exists l, r.\ \gamma(\mathtt{x}) = (0, l, r)\}$
// $\{\mathsf{graph}(\mathtt{x},\gamma) \wedge \gamma(\mathtt{x}) = (0, l, r)\}$
// $\{\mathsf{vertices\_at}(\mathsf{reachable}(\gamma, \mathtt{x}), \gamma) \wedge \gamma(\mathtt{x}) = (0, l, r)\}$
// $\{\mathsf{vertices\_at}(R, \gamma) \wedge \gamma(\mathtt{x}) = (0, l, r)\}$
// $\searrow \{\mathtt{x} \mapsto 0, l, r \wedge \gamma(\mathtt{x}) = (0, l, r)\}$
    l = x -> l;
    r = x -> r;
    x -> m = 1;
// $\swarrow \{\mathtt{x} \mapsto 1, \mathtt{l}, \mathtt{r} \wedge \gamma(\mathtt{x}) = (0, \mathtt{l}, \mathtt{r}) \wedge \exists \gamma_1.\ \mathit{mark1}(\gamma, \mathtt{x}, \gamma_1)\}$
// $\{\exists \gamma_1.\ \mathsf{vertices\_at}(R, \gamma_1) \wedge \gamma(\mathtt{x}) = (0, \mathtt{l}, \mathtt{r}) \wedge \mathit{mark1}(\gamma, \mathtt{x}, \gamma_1)\}$
// $\{\mathsf{vertices\_at}(R, \gamma_1) \wedge \gamma(\mathtt{x}) = (0, \mathtt{l}, \mathtt{r}) \wedge \mathit{mark1}(\gamma, \mathtt{x}, \gamma_1)\}$
    if (l) {
        root_mark = l -> m;
        if (root_mark == 0) {
            spanning(l);
        } else { x -> l = 0; } }
// $\left\{\begin{array}{l}\exists \gamma_2.\ \mathsf{vertices\_at}(R, \gamma_2) \wedge \gamma(\mathtt{x}) = (0, \mathtt{l}, \mathtt{r}) \wedge \\ \mathit{mark1}(\gamma, \mathtt{x}, \gamma_1) \wedge \mathit{e\_span}(\gamma_1, \mathtt{x.L}, \gamma_2)\end{array}\right\}$
// $\left\{\begin{array}{l}\mathsf{vertices\_at}(R, \gamma_2) \wedge \gamma(\mathtt{x}) = (0, \mathtt{l}, \mathtt{r}) \wedge \\ \mathit{mark1}(\gamma, \mathtt{x}, \gamma_1) \wedge \mathit{e\_span}(\gamma_1, \mathtt{x.L}, \gamma_2)\end{array}\right\}$
    if (r) {
        root_mark = r -> m;
        if (root_mark == 0) {
           spanning(r);
        } else { x -> r = 0; } }
// $\left\{\begin{array}{l}\exists \gamma_3.\ \mathsf{vertices\_at}(R, \gamma_3) \wedge \gamma(\mathtt{x}) = (0, \mathtt{l}, \mathtt{r}) \wedge \\ \mathit{mark1}(\gamma, \mathtt{x}, \gamma_1) \wedge \mathit{e\_span}(\gamma_1, \mathtt{x.L}, \gamma_2) \wedge \mathit{e\_span}(\gamma_2, \mathtt{x.R}, \gamma_3)\end{array}\right\}$
} // $\{\exists \gamma_3.\ \mathsf{vertex\_at}(\mathsf{reachable}(\gamma, \mathtt{x}), \gamma_3) \wedge \mathit{span}(\gamma, \mathtt{x}, \gamma_3)\}$
```

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

$$\begin{aligned}\mathit{span}(\gamma_1, \mathrm{x}, \gamma_2) \triangleq\ & \mathit{mark}(\gamma_1, \mathrm{x}, \gamma_2) \wedge \gamma_1 \uparrow (\lambda v. x \overset{\gamma_1}{\leadsto}{}^{\star}_{0}\ v) \text{ is a tree} \\ & \gamma_1 \uparrow (\lambda v. \neg x \overset{\gamma_1}{\leadsto}{}^{\star}_{0}\ v) = \gamma_2 \uparrow (\lambda v. \neg x \overset{\gamma_1}{\leadsto}{}^{\star}_{0}\ v) \wedge \\ & (\forall v.\ x \overset{\gamma_1}{\leadsto}{}^{\star}_{0}\ v \Rightarrow \gamma_2 \models x \leadsto v) \wedge \\ & (\forall a, b.\ x \overset{\gamma_1}{\leadsto}{}^{\star}_{0}\ a \Rightarrow \neg x \overset{\gamma_1}{\leadsto}{}^{\star}_{0}\ b \Rightarrow \neg \gamma_2 \models a \leadsto b)\end{aligned}$$

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

Figure 10. Clight code and proof sketch for bigraph spanning tree.

of an algorithm since its focus on heavier automation tends to trade off proving very exact specifications (*e.g.* H/S proofs about lists usually treat their values as multisets instead of sequences). We do not give H/S any "hints" at intermediate program points.

Line 1 defines the recursive data structure `node` and lines 19–21 use H/S's normal user-defined recursive predicate feature to define the `graph` predicate. H/S uses the keyword `self` to refer to the object being defined; in other words H/S's definition is very close to the fold/unfold relationship given as equation (1) in Figure 1 and we can justify its soundness as in §4.2. H/S uses the `::` operator for $\mapsto$ and to provide the root pointer for a recursive predicate, `U*` for $\uplus$ and automatically quantifies free variables existentially. H/S

```
1  data node { int val; node left; node right; }
2  
3  relation lookup(abstract G, node x,
4    int d, node l, node r).
5  relation update(abstract G, node x, int d,
6    abstract G1).
7  relation mark(abstract G, node x, abstract G1).
8  relation
9    subset_reach(abstract G, node x, abstract G1).
10 relation
11  eq_notreach(abstract G, node x, abstract G1).
12 
13 axiom lookup(G,x,1,l,r) ==> mark(G,x,G).
14 axiom mark(G,x,G1) & lookup(G,y,v,l,r) ==>
15   subset_reach(G,x,G1) & eq_notreach(G,x,G1) &
16   lookup(G1,y,_,l,r).
17 // ... other axioms elided ...
18 
19 graph<G> == self = null or
20  self::node<v,l,r> U* l::graph<G> U* r::graph<G>
21  & lookup(G,self,v,l,r);
22 
23 rlemma "subgraphupdate_l" l::graph<G1> *
24  (l::graph<G> --@ (x::node<v,l,r> U*
25  (l::graph<G> U* r::graph<G>))) &
26  subset_reach(G,l,G1) & eq_notreach(G,l,G1)
27  & lookup(G,x,v,l,r) & lookup(G1,x,v1,l,r)
28   -> x::node<v1,l,r> U*
29     (l::graph<G1> U* r::graph<G1>);
30 // ... other ramification lemmas elided ...
31 
32 void mark(node x)
33 requires x::graph<G>
34 ensures x::graph<G1> & mark(G,x,G1); {
35   node l, r;
36   if (x == null) return;
37   else {
38     if (x.val == 1) return;
39     l = x.left;
40     r = x.right;
41     x.val = 1;
42     mark(l);
43     mark(r);
44 } }
```

Figure 11. Bigraph marking in HIP/SLEEK

does not need alignment restrictions because it enjoys a Java-like memory model with objects and fields: *i.e.* it is not possible to have a pointer pointing into the "middle" of a record. The last line of the `graph` predicate definition (21) includes the `lookup` abstract relation. In Figure 1, this would be written $\gamma(\mathtt{self}) = (v, l, r)$.

6.2 Externally-verified lemmas

To implement ramifications in HIP/SLEEK we extended its lemma system (Nguyen and Chin 2008). Normal lemmas in H/S are user defined, automatically checked, and automatically applied in program verifications. In contrast, our new lemmas are still user defined and still automatically applied in program verifications, but **not** automatically checked, so we call them *externally verified lemmas*; their key advantage is that they can be much more complex than the lemmas H/S can check automatically. Instead of checking the lemmas automatically, H/S outputs a Coq `Module Type` that states the lemmas H/S needed in the verification. Users then implement a matching `Module` to get a fully-verified result.

The first step to adding more general kinds of lemmas is to add a notion of abstract relations; the relations used by the `mark` verification are given in lines 3–11. We have already met `lookup`;

lines 5 and 7 are how H/S models *mark1* and *mark*, respectively. Users do not provide any definitions for these relations, but we do give H/S some axioms for how they behave: *e.g.* line 13 contains tells H/S that if the root of a graph is marked then we can consider the whole graph marked. This axiom is used on line 38 to safely `return` once we encounter an already-marked `node`. Users also provide spatial ramification rules as in lines 23–29.

6.3 Automatic ramification

HIP/SLEEK reasons about graph algorithms using three key ideas. First, H/S uses fold/unfold relationships whenever it needs to reason about a recursive predicate during the entailment checking that occurs during forward reasoning. For example, to check the dereference in line 38, the `x::graph<G>` predicate is unfolded to reach[4]

$$\exists m,l,r.\ \gamma(\mathtt{x}) = (m,l,r) \wedge \mathtt{x} \mapsto m,l,r \uplus \mathsf{graph}(l,\gamma) \uplus \mathsf{graph}(r,\gamma)$$

Second, once HIP/SLEEK realizes that its forward reasoning needs to isolate a predicate that is $\uplus$-shared, it uses the existential wand "septraction" operator $-\!\!\circledast$ (defined in Figure 2) to reach the strongest post condition. The existential wand $-\!\!\circledast$ is simpler to introduce than the universal one $-\!\!*$ during forward reasoning because H/S already knows G_1 and L_1 when it needs to calculate a residual frame R. For example, the precondition at line 42 is

$$\gamma(\mathtt{x}) = (0,\mathtt{l},\mathtt{r}) \wedge \mathit{mark1}(\gamma,\mathtt{x},\gamma') \wedge \\ \mathtt{x} \mapsto 1,\mathtt{l},\mathtt{r} \uplus \mathsf{graph}(\mathtt{l},\gamma') \uplus \mathsf{graph}(\mathtt{r},\gamma')$$

and since H/S knows the call to `mark(l)` requires $\mathsf{graph}(\mathtt{l},\gamma')$, it can calculate the frame residue R as $Q -\!\!\circledast P$ since it knows that[5]

$$\overbrace{\mathtt{x} \mapsto 1,\mathtt{l},\mathtt{r} \uplus \mathsf{graph}(\mathtt{l},\gamma') \uplus \mathsf{graph}(\mathtt{r},\gamma')}^{P} \vdash \overbrace{\mathsf{graph}(\mathtt{l},\gamma')}^{Q} * \\ \underbrace{\Big(\mathsf{graph}(\mathtt{l},\gamma') -\!\!\circledast \big(\mathtt{x} \mapsto 1,\mathtt{l},\mathtt{r} \uplus \mathsf{graph}(\mathtt{l},\gamma') \uplus \mathsf{graph}(\mathtt{r},\gamma')\big)\Big)}_{R}$$

Note that in general $P \vdash Q * (Q -\!\!\circledast P)$ is **not** a tautology since not every P has a Q "hiding inside it". However, H/S's unfold of graph makes it readily apparent that $\mathsf{graph}(l,\gamma)$ **is** inside the premise of the entailment P, so H/S is justified in calculating the frame R.

To calculate the postcondition of line 42, H/S takes the residue R and $*$-combines it with the postcondition of the `mark(l)` call, *i.e.*:

$$\big(\mathsf{graph}(\mathtt{l},\gamma'') \wedge \mathit{mark}(\gamma',\mathtt{l},\gamma'')\big) * \gamma(\mathtt{x}) = (0,\mathtt{l},\mathtt{r}) \wedge \mathit{mark1}(\gamma,\mathtt{x},\gamma') \\ \wedge \Big(\mathsf{graph}(l,\gamma') -\!\!\circledast \big(x \mapsto 1,\mathtt{l},\mathtt{r} \uplus \mathsf{graph}(\mathtt{l},\gamma') \uplus \mathsf{graph}(\mathtt{r},\gamma')\big)\Big)$$

The third and final step is to use lemmas to eliminate the $-\!\!\circledast$ operator. After line 42 H/S uses the lemma from lines 23–29:

$$\bigg(\Big(\mathsf{graph}(l,\gamma) -\!\!\circledast \big(x \mapsto m,l,r \uplus \mathsf{graph}(l,\gamma) \uplus \mathsf{graph}(r,\gamma)\big)\Big) \\ * \mathsf{graph}(l,\gamma') \wedge \mathit{subset\_reach}(\gamma,l,\gamma') \wedge \mathit{eqnot\_reach}(\gamma,l,\gamma') \\ \wedge \gamma(x) = (m,l,r) \wedge \gamma'(x) = (m1,l,r)\bigg) \Rightarrow \\ \big(x \mapsto m1,l,r \uplus \mathsf{graph}(l,\gamma') \uplus \mathsf{graph}(r,\gamma')\big)$$

Note that this lemma does not mention *mark* explicitly, allowing it to be reused in other algorithms. The connection between the *mark*, *subset_reach*, and *eqnot_reach* relations is given by the axiom on lines 14–16; other (elided) lemmas connect *mark1* as well. H/S must apply all of them, guided by structural matching, to eliminate the $-\!\!\circledast$ to reach the following relatively pleasant-looking

[4] We will write predicates in a more conventional mathematical style rather than the ASCII format used by H/S.

[5] We omit pure conjuncts *e.g.* $\mathit{mark1}(\gamma,\mathtt{x},\gamma')$ from both P and R.

```
Module Type Mgraphmark.
 ...
 Parameter G : Type.
 Parameter node : Type.
 Parameter graph : node -> G -> formula.
 Parameter mark : G -> node -> G -> formula.
 ...
 Axiom axiom_3 : forall l r x G,
   valid (imp (lookup G x true l r) (mark G x G)).
 ...
 Axiom subgraphupdate_l : forall G v G1 x v1 l r,
  valid (imp (and (star (graph l G1)
  (mwand (graph l G) (union (ptto_node x v l r)
  (union (graph l G) (graph r G)))))
  (and (subset_reach G l G1) (and
  (eq_notreach G l G1) (and (lookup G x v l r)
  (lookup G1 x v1 l r)))))
  (union (ptto_node x v1 l r) (union (graph l G1)
  (graph r G1)))).
 ...
End Mgraphmark.
```

Figure 12. Coq `Module Type` generated by HIP/SLEEK

postcondition of line 42

$$\gamma(\mathtt{x}) = (0,\mathtt{l},\mathtt{r}) \wedge \mathit{mark1}(\gamma,\mathtt{x},\gamma') \wedge \mathit{mark}(\gamma',\mathtt{l},\gamma'') \wedge \\ \mathtt{x} \mapsto 1,\mathtt{l},\mathtt{r} \uplus \mathsf{graph}(\mathtt{l},\gamma'') \uplus \mathsf{graph}(\mathtt{r},\gamma'')$$

H/S now continues the verification with the next command `mark(r)`.

6.4 Generating the Coq module type

The lemmas and axioms that HIP/SLEEK has used in the verification still need to be checked. Accordingly, HIP/SLEEK generates a Coq file with a `Module Type` specifying them. The `Module Type` generated for `mark` is given in Figure 12 to illustrate this process. To obtain a complete verification for `mark`, users must build a Coq `Module` that satisfies `Mgraphmark` using definitions for the abstract relations that they think are reasonable (in particular, for the `mark` relation, since it is used in the specification of the algorithm).

6.5 Consequences of HIP/SLEEK's style of reasoning

HIP/SLEEK's style of reasoning has a few consequences for how we proceed. First, H/S's use of the fold/unfold relationship means that we need the "supplementary" package from the ramification library (§4.3) to enable it. Second, H/S's use of the existential wand $-\!\!\circledast$ means we need to convert our $-\!\!*$ ramification lemmas to contain $-\!\!\circledast$. Fortunately this is easy, assuming L_1 is precise:

$$\textsc{WandToEwand}\quad \frac{G_1 \vdash L_1 * (L_2 -\!\!* G_2)}{(L_1 -\!\!\circledast G_1) * L_2 \vdash G_2}\ \mathit{precise}(L_1)$$

The supplementary package proves that the graph predicate is "precise", so all of our graph ramification lemmas work without additional effort. Accordingly, to provide the `Module` implementing `Mgraphmark` not burdensome, especially starting from a working Floyd proof that uses the same mathematical relations.

7. Statistics related to our development

All of our results in this paper have been machine-checked. The vast bulk of our development was checked in Coq, although a 55-line file (shown in Figure 11) was checked in HIP/SLEEK. Our modifications to HIP/SLEEK itself were not machine-checked since HIP/SLEEK does not have a mechanized soundness proof. Although the size of a development does not perfectly match with

[6] H/S files modified here is not necessarily fresh created.

Component	Number of files	Size (in lines)
Math Graph (§3)	19	12,628
Spatial Graph ($4)	12	7,337
Integration into Floyd (§5)	12	1,917
Modifications to H/S (§6)	51[6]	2,500
VST Examples (§5.2)	13	3,253
H/S Example (§6)	2	429
Memory Model & Logic	13	2,395
Common Utilities	10	3,085
Total Development	132	33,544

Table 1. Size of our codebase

that development's importance or implementation difficulty, we present it nonetheless in Table 1, organized roughly by the paper section corresponding to each development. It took fewer than 400 lines of Coq to verify the `mark` algorithm in HIP/SLEEK, indicating that a large portion of the codebase is shared between VST and H/S.

8. Related work

Comparison with Hobor and Villard. The most direct ancestor of our work is (Hobor and Villard 2013), which focused on verifying graph algorithms using and introduced the RAMIFY rule. We have generalized this rule to better handle modified program variables and existential quantifiers in postconditions; they hacked their way around these issues by proposing a 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. They proposed no way to verify unmodified program code, to modify program variables inside nested localization blocks, or to do a ramification across multiple assignments as we do in lines 16–20 of figure 1. Hobor and Villard avoided existentials in localized postconditions because they defined all of their mathematical operations (*e.g. mark*, *mark1*) as functions rather than as relations.

Hobor and Villard treated mathematical graphs very simply, as triples (V, E, L) of vertices, edges, and a labeling function on vertices. Vertices had no more than two neighbors. In contrast, our mathematical graph framework (§3) is very modular and general and has been tuned to work smoothly in a mechanized context.

Hobor and Villard fell into the trap of defining spatial graphs recursively (§4.1); unfortunately other members of the research community have since followed them in. We exposed this error and provided a sound and quite general definition for `graph` (§4.2) that recovers fold/unfold reasoning. We developed a much more general and more modular set of related lemmas and connect our spatial reasoning to two very different verification tools (§4.3), VST (§5) and HIP/SLEEK (§6). Our development is entirely machine-checked (§7) whereas they used only pen and paper.

Other verification of graph algorithms, with or without $\uplus$. Yang's verification of the Schorr-Waite algorithm is a landmark in the early separation logic literature (Yang 2001). Bornat *et al.* gave an early attempt to reason about graph algorithms in separation logic in a more general way (Bornat et al. 2004).

Reynolds was the first to document the overlapping conjunction $\uplus$, although he did not present any strategy to reason about it using Hoare rules (Reynolds 2003). Gardner *et al.* were the first to reason about a program using $\uplus$ in Javascript (Gardner et al. 2012). Raad *et al.* used $\uplus$ to reason about a concurrent spanning algorithm using a kind of "concurrent localization" (Raad et al. 2015). Sergey *et al.* also verified a concurrent spanning tree algorithm and mechanized their proofs in Coq (Sergey et al. 2015).

Almost a decade after Yang verified Schorr-Waite on paper, Dafny automated its verification (Leino 2010).

Local variables. An alternative way to avoid local variable issues is to use "variables as resource" (Bornat et al. 2006). However, most mechanized verification systems do not use variables as resource (Beckert et al. 2007; Distefano and Parkinson 2008; Chin et al. 2010; Leino 2010; Bengtson et al. 2012; Appel et al. 2014).

Verification tools. Our work heavily interacts with the Floyd (Appel et al. 2014) and HIP/SLEEK (Chin et al. 2010) verification tools. Like Floyd, Charge! 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). A more automated approach to verification of low level programs using Coq is implemented in the Bedrock framework (Chlipala 2011).

Many 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 Parkinson 2008), and Verifast (Jacobs et al. 2011). One of 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 KeY (Beckert et al. 2007) are two notable verifiers not based on separation logic; KeY uses an interactive verifier while Dafny pursues more automation by using the SMT solver Z3 (de Moura and Bjørner 2008).

Mechanized mathematical graph theory. There is a long history, going back at least 25 years, of mechanized reasoning about mathematical graphs (Wong 1991). The most famous mechanically verified "graph theorem" is the Four Color Theorem (Gonthier 2005); however the development actually uses hypermaps instead of graphs. Noschinski built a graph library in Isabelle/HOL whose formalization is the most similar to ours (Noschinski 2015a), *e.g.* supporting graphs with labeled and parallel arcs.

Noschinski and Dubois *et al.* used proof assistants to design verifiable checkers for solutions to graph problems (Noschinski 2015b; Dubois et al. 2015). Yamamoto *et al.* and Bauer and Nipkow use an alternative inductive encoding of graphs to formalize planar graph theory (Yamamoto et al. 1995; Bauer and Nipkow 2002).

9. Future work and conclusion

In the future we plan to improve the pure reasoning of graphs and similar data structures. We plan to verify a garbage collector algorithm for the "CertiCoq" project, which is building a certified compiler from Gallina to Clight. We would like to investigate using our externally verified lemmas in HIP/SLEEK to verify code such as fast exponentiation and more graph algorithms. We also would like to make the interface between Coq and HIP/SLEEK simpler and cleaner. One final direction we would like to investigate is using our new connection to Coq to have HIP/SLEEK output certificates as it verifies programs so that the system becomes more trustworthy.

Our main contributions were as follows. We generalized the RAMIFY rule to handle modified program and existential quantifiers in postconditions more smoothly. We developed a general and modular framework for reasoning about mathematical graphs and a general and modular spatial library for reasoning about graphs in the heap. We provided a sound definition for `graph` that still obeyed the fold/unfold relationship. We connected our reasoning to two verification tools and used them to verify several graph-manipulating algorithms.

Acknowledgements. This material was based in part on research supported by Yale-NUS College and R-607-265-045-121. All opinions expressed in this work are solely those of the authors.

This material is based in part on research sponsored by DARPA under agreement number FA8750-12-2-0293. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

References

A. W. Appel and D. McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Transactions on Programming Languages and Systems*, 23(5):657–683, 2001.

A. W. Appel, P.-A. Melliès, C. D. Richards, and J. Vouillon. 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)*, pages 109–122, Jan. 2007.

A. W. Appel, R. Dockins, A. Hobor, L. Beringer, J. Dodds, G. Stewart, S. Blazy, and X. Leroy. *Program Logics for Certified Compilers*. Cambridge University Press, New York, NY, USA, 2014. ISBN 110704801X, 9781107048010.

G. Bauer and T. Nipkow. The 5 colour theorem in isabelle/isar. In *International Conference on Theorem Proving in Higher Order Logics*, pages 67–82. Springer, 2002.

B. Beckert, R. Hähnle, and P. H. Schmitt. *Verification of Object-oriented Software: The KeY Approach*. Springer-Verlag, Berlin, Heidelberg, 2007. ISBN 3-540-68977-X, 978-3-540-68977-5.

J. Bengtson, J. B. Jensen, and L. Birkedal. 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*, pages 315–331, 2012.

J. Berdine, C. Calcagno, and P. W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO*, pages 115–137, 2005.

S. Blazy and X. Leroy. Mechanized semantics for the clight subset of the C language. *J. Autom. Reasoning*, 43(3):263–288, 2009.

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

R. Bornat, C. Calcagno, and H. Yang. Variables as resource in separation logic. *ENTCS*, 155:247–276, 2006.

W. N. Chin, C. David, H. H. Nguyen, and S. Qin. Automated verification of shape, size and bag properties via user-defined predicates in separation logic. *Science of Computer Programming*, 77(9):1,006–1,036, 2010.

A. Chlipala. 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*, pages 234–245, 2011.

A. Chlipala. *Certified Programming with Dependent Types - A Pragmatic Introduction to the Coq Proof Assistant*. MIT Press, 2013. ISBN 978-0-262-02665-9. URL http://mitpress.mit.edu/books/certified-programming-dependent-types.

L. M. de Moura and N. Bjørner. 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*, pages 337–340, 2008. doi: 10.1007/978-3-540-78800-3_24. URL http://dx.doi.org/10.1007/978-3-540-78800-3_24.

D. Distefano and M. J. Parkinson. 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*, pages 213–226, 2008. doi: 10.1145/1449764.1449782. URL http://doi.acm.org/10.1145/1449764.1449782.

C. Dubois, S. Elloumi, B. Robillard, and C. Vincent. Graphes et couplages en coq. In *Vingt-sixièmes Journées Francophones des Langages Applicatifs (JFLA 2015)*, 2015.

R. W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Proceedings of the Symposium on Applied Mathematics*, volume 19, pages 19–32. AMS, 1967.

P. Gardner, S. Maffeis, and G. D. Smith. 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*, pages 31–44, 2012. doi: 10.1145/2103656.2103663. URL http://doi.acm.org/10.1145/2103656.2103663.

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

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

B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. 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*, pages 41–55, 2011.

K. R. M. Leino. 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*, pages 348–370, 2010. doi: 10.1007/978-3-642-17511-4_20. URL http://dx.doi.org/10.1007/978-3-642-17511-4_20.

X. Leroy. 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*, pages 42–54, 2006.

H. H. Nguyen and W. Chin. Enhancing program verification with lemmas. In *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings*, pages 355–369, 2008. doi: 10.1007/978-3-540-70545-1_34. URL http://dx.doi.org/10.1007/978-3-540-70545-1_34.

L. Noschinski. A graph library for isabelle. *Mathematics in Computer Science*, 9(1):23–39, 2015a. ISSN 1661-8289. doi: 10.1007/s11786-014-0183-z. URL http://dx.doi.org/10.1007/s11786-014-0183-z.

L. Noschinski. *Formalizing Graph Theory and Planarity Certificates*. PhD thesis, Universität München, 2015b.

A. Raad, J. Villard, and P. Gardner. 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*, pages 710–735, 2015. doi: 10.1007/978-3-662-46669-8_29. URL http://dx.doi.org/10.1007/978-3-662-46669-8_29.

J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74, 2002.

J. C. Reynolds. A short course on separation logic. http://www.cs.cmu.edu/afs/cs.cmu.edu/project/fox-19/member/jcr/wwwaac2003/notes7.ps, 2003.

I. Sergey, A. Nanevski, and A. Banerjee. Mechanized verification of fine-grained concurrent programs. In *PLDI*, pages 77–87, 2015.

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

W. Wong. A simple graph theory and its application in railway signaling. In *HOL Theorem Proving System and Its Applications, 1991., International Workshop on the*, pages 395–409, Aug 1991. doi: 10.1109/HOL.1991.596304.

M. Yamamoto, S.-y. Nishizaki, M. Hagiya, and Y. Toda. Formalization of planar graphs. In *International Conference on Theorem Proving in Higher Order Logics*, pages 369–384. Springer, 1995.

H. Yang. *Local Reasoning for Stateful Programs*. PhD thesis, University of Illinois, 2001.