The Ramifications of Mechanized Localizations within Data Structures

Abstract

We show how to mechanically verify programs manipulating data structures with intrinsic sharing such as heap-represented graphs. We upgrade the 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 for graphs in separation logic. We connect our theories to two verification tools with different levels of automation and use them to verify several canonical graph algorithms. Our proofs are entirely machine-checked in Coq.

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 CITE CITE CITE. Unfortunately, verifying programs that manipulate graph-like data structures (e.g. structures with *intrinsic sharing*) has been far more challenging. Indeed, verifying such programs was formidable enough that a number of the early landmark results in separation logic devoted substantial efforts to verify single examples such as Schorr-Waite [Yang 2001] and XXX CITE with pen and paper—avoiding entirely the additional challenges inherent in machine-assisted 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 with pen and paper [Hobor and Villard 2013], but left open the question of how such proofs could be incorporated in a machine-assisted setting. In this paper, we show how this can be done, and demonstrate the value of our approach by adding ramification to two rather sizeable—albeit quite differently flavored—separation logic-based verification tools: the Coq-based tactic system of the Verified Software Toolchain CITE and the more highly-automated HIP/SLEEK program verifier [Chin et al. 2010]. Despite the substantial differences between these systems, the vast majority of our infrastructure is shared between them, and since many of the other computer-assisted verification tools under development today CITE CITE CITE CITE have much in common with at least one of these tools, we believe that our techniques will be applicable to many other systems as well.

Along the way we develop an improved proof rule for ramification that supports existential variables and enjoys a smoother interaction with

with modified

references to modified local program variables more generally.

smoothly

a smoother and more uniform support for modified program variables.

along with a more

make a number upgrades to the theory of ramification and present

, with a better treatment of modified variables, existentials,

Along the way we discover—and show how to fix—a rather subtle error in Hobor and Villard's presentation: neither the Knaster-Tarski [Tarski 1955] nor the Appel-McAllester [Appel and McAllester 2001] method for solving recursive fixpoints is suitable for defining recursive graph predicates in separation logic.

We also develop a general framework for defining and reasoning about mathematical graphs and

different kinds of mathematical graphs can be implemented in separation logic in a uniform way;

generalize their setting so that it can handle a wider variety of data structures;

We use both systems to verify a number of different programs utilizing graph-manipulating structures, letting us understand the advantages and disadvantages of both.

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

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

Contributions and structure of the remainder of this paper.

- Example §**??**. Contributions: new ramify rules, new notation, everything machine-checked, multiple tools sharing mathematical infrastructure.
- Mathematical graphs. Contributions: computable, compositional, & general graph library in Coq. Treatment of null.
- Spatial graphs. Contributions: correct general graph predicate. Problem with fixed point. Problem with "later" not being precise. Fold/unfold, precise, etc.
- Integrating ramification into verification tools. Contributions: VST (localize/unlocalize). H/S (external axioms). Additional examples. New proof of "copy" that does not use regions.
- Related work, future work, and conclusion

2. Generalizing 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 outline of our verification sketch draws heavily upon the first ramification example done by Hobor and Villard [Hobor and Villard 2013] but we have a number of improvements as outlined below.

First, the code in Figure 1 is written in the Clight language [Blazy and Leroy 2009]. Clight is an input language to the CompCert certified compiler [Leroy 2006], which compiles our code exactly as written. **Second**, the paper-format verification sketch for `mark` in Figure 1 is backed by a fully machine-checked proof using an upgraded version of the Floyd system of the Verified Software Toolchain [Appel et al. 2014]; our upgrades are explained in §6. Accordingly, there is an unbroken certified chain from our specification of `mark` all the way to the assembly code. The program invariants in Figure 1 are almost exactly what is used in the Floyd proof, with only minor cleanup for paper-based presentation.

The specification we certify (lines 6 and 33) is

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

The specification is for full functional correctness and stated incorporating some *mathematical* graphs γ. Our **third** improvement on Hobor and Villard is that the definition of γ we employ is quite general; we defer it until §3. For now consider γ to be a function that maps a vertex $v \in V$ to triples (m,l,r), where m is a "color" bit (0 or 1) and $\{l,r\} \subseteq V \uplus \{0\}$ are the neighbors of v. Neighbors can take a non-V "null" value.

The graph predicate is *spatial*, *i.e.* defined with separation logic operators such as maps-to $\vdash>$, and explains how the mathematical graph γ is actually implemented in the heap. Our **fourth** improvement on Hobor and Villard is that our spatial graph predicate is also quite general, so we defer it until §**??**. For now it is enough to know that graph satisfies the following fold/unfold relationship:

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

This fold/unfold relationship has several interesting points. First, as we explain in §**??**, it is a terrible mistake to write (1) as a definition using $\triangleq$ rather than as a biimplication using $\iff$. Second, as in Hobor and Villard, (1) uses the less-standard "overlapping conjunction" $\circledast$ of separation logic. The semantics of $\sigma \models P \circledast Q$ is that the state σ divides into *three* disjoint parts σ_1, σ_2, and σ_3 such that $\sigma_1 \oplus \sigma_2 \models P$ and $\sigma_2 \oplus \sigma_3 \models Q$. That is, P and Q share the existentially-quantified substate σ_2, which we use in the unfolded side of (1) to indicate that all portions of the graph can overlap (*i.e.*, nodes in the left subgraph can also be in the right subgraph or even be the root x). Third (1) includes an example of a consequence of how an industrial-strength setting complicates verification. Lines 1–4 define the data type `Node` used by `mark`. The `_Allignas(n)` directives force the compiler to align fields on n-byte boundaries. As explained in §**??**, this alignment is necessary in C-like memory models to prove the fold-unfold relationship (1), which is why (1) includes an alignment restriction $x \bmod 16 = 0$ and an existentially-quantified "blank" second field for the root $x \mapsto m, -, l, r$. (In our Floyd proofs the alignment restriction and blank second field are nicely hidden "behind the scenes", but they are there if you unfold the definitions enough.)

Our **fifth** improvement on Hobor and Villard is that the postcondition of `mark` is specified *relationally*, *i.e.*

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

instead of *functionally*, *i.e.*

$$\{\mathsf{graph}\big(\mathtt{x}, \mathit{mark}(\gamma, \mathtt{x})\big)\}$$

In both cases a mathematical definition mark is used; in the first case it is a relation that specifies that γ' is the result of correctly marking γ from $\mathtt{x}$, whereas in the second it is a function that computes the result of marking γ from $\mathtt{x}$. For both practical and theoretical reasons a relational approach is better. Practically, it is remarkably painful to define computational functions over graphs in a proof assistant like Coq. Substantial portions of this pain are even unnecessary: 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. In contrast, defining relations is much easier because *e.g.* one can use quantifiers and do not have to prove termination. Theoretically, relations are preferable because they are more general (*e.g.* they allow some "inputs" to not have "outputs", *i.e.* be partial; or conversely allow multiple outputs from a single input, *i.e.* in nondeterministic settings) and more compositional (*i.e.*, we can reuse relations and associated lemmas in other verifications). We take advantage of compositionality by using $\mathit{mark}(\gamma, x, \gamma') /\!\!| \ldots$ to specify both our "spanning tree" and "graph copy" algorithms, which also mark nodes while carrying out their primary task.

The actual definition of the mark relation is straightforward. Our mathematical graphs admit natural definitions of reachability, and so $\mathit{mark}(\gamma, x, \gamma')$ means that

$$\forall v.\, \gamma'(v) = \begin{cases} (1, l, r) & \text{when } x \overset{\gamma}{\rightsquigarrow}{}^{*}_{0} v \text{ and } \gamma(v) = (0, l, r) \\ \gamma(v) & \text{otherwise} \end{cases}$$

That is, γ and γ' have the same vertices and edges but the mark bits in γ' have been set to 1 (marked) in all nodes x' reachable from x in γ along an unmarked path. Note the use of the $\forall$ quantifier, permitted because mark is a relation.

Turning to the body of the verification (lines 7–32), our **sixth** improvement over Hobor and Villard is perhaps the most readily apparent visually: blocks of proof script bracketed by the symbols ↘ and ↙, such as lines 11–13. We call a bracketed set of lines like this a "localization block". The intuitive idea is that we zoom in from a larger "global" context into a smaller "local" one. After verifying some commands locally to arrive at the local postcondition we zoom back out to the global postcondition. In the case of these lines, you can imagine unfolding the graph predicate in line 10 using equation (1) and then zooming in to the root node `x` for lines 11–13, before zooming back out in line 14. Although we do not use this power here, localization blocks can be safely nested. We will define localization blocks formally in §2.1. This notation was inspired by the way we mechanized our theories as explained in §6, but we believe it to be useful outside of mechanized contexts.

Program variables §2.2, existentials §2.3 4.2. Using existential quantifier in post condition.

Also in H/S **??**

Additional comparison of our solution to Hobor and Villard's is given in §9.

2.1 Frames and ramifications are localizations

The key rule of separation logic is FRAME [Reynolds 2002]:

$$\frac{\{P\}\; c\; \{Q\}}{\{P * F\}\; c\; \{Q * F\}}\ \textsc{Frame} \qquad F \text{ IGNORES } \mathsf{ModVar}(c)$$

The 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:

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

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

The reason why FRAME is so important is because it enables local verifications. That is, rather than BLAH.

Hobor and Villard observed that FRAME is bit rigid because it forces verifiers to split program assertions into syntactically $*$-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:

$$\textsc{Ramify}\qquad \frac{\{L_1\}\ c\ \{L_2\} \qquad G_1 \vdash L_1 * (L_2 \mathrel{-\,-\,*} G_2)}{\{G_1\}\ c\ \{G_2\}} \quad \begin{array}{c}(L_2 \mathrel{-\,-\,*} G_2)\\ \text{IGNORES}\\ \mathsf{ModVar}(c)\end{array}$$

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* $G_1 \vdash L_1 * (L_2 \mathrel{-\,-\,*} G_2)$. Essentially the ramification entailment ensures that the change in state specified locally fits properly into the global context.

The Ramify rule is sound but interacts poorly with modified program variables (as in lines 17–21 of Figure 1) and localized existentials (as in lines 23–27). Both of these limitations are annoying enough in paper proofs and graduate to major headaches in mechanized ones. Happily, we show how to overcome both limitations in §2.2 and §2.3, respectively.

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 `$(i)$     $c_1$;  ...  ;  $c_n$;
4 // $\swarrow \{L_2\}$
5 // $\{G_2\}$
```

we mean apply the Ramify rule with $G_1 \vdash L_1 * (L_2 \mathrel{-\!*} G_2)$. An advantage of this notation is crystal clarity on the predicates used in the ramification entailment. For convenience, the optional `(i) specification can reference an equation or lemma number that solves the ramification entailment. If we wish to save a line or two we can compress *e.g.* 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.

In §2.2 and §2.3 we present new variants of Ramify. Our notation carries over without any meaningful change: just use the new rules to enable the more general ramification entailments they permit. When in doubt the most general rule, Ramify-PQ from §2.3, implies all of the others.

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:

```
// $\{P_1 * F\}$
   $c_1$;
// $\{P_2 * \ldots\}$
   $c_2$;
// $\{P_3 * \ldots\}$
   $c_3$;
// $\{P_4 * F\}$
```

New notation:

```
// $\{P_1 * F\} \searrow \{P_1\}$
         $c_1$;
//       $\{P_2\}$
         $c_2$;
//       $\{P_3\}$
         $c_3$;
// $\{P_4 * F\} \swarrow \{P_4\}$
```

2.2 The program variable bugaboo

Intro blah. lines 23–27 of Figure 1

Consider using ramification to verify the following program:

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

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

$$(x = 5/|A) \vdash (x = 5/|B) * \big((x = 6/|C) \mathrel{-\!*} (x = 6/|D)\big)$$

One problem is that $(x = 6/|C) \mathrel{-\!*} (x = 6/|D)$ does **not** ignore the modified program variable x, preventing us from applying the Ramify rule. Intuitively, the stability 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.

The obvious thing to try is 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:

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

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 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:

$$(x = 5/|A) \vdash (x = 5/|B) * \\ [\![\texttt{...; x = x + 1; ...;}]\!]\big((x = 6/|C) \mathrel{-\!*} (x = 6/|D)\big)$$

Since x is modified, its value can change from the first line, in which x must be 5, to the second, in which 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.

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:

SOLVE RAMIFY-P

$$\frac{G_1 \vdash L_1 * F \qquad F \vdash L_2 -\!* G_2}{G_1 \vdash L_1 * [\![c]\!](L_2 -\!* G_2)} \; F \text{ IGNORES } \mathsf{ModVar}(c)$$

That is, 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. To connect the pair, we must choose a suitable predicate F that ignores modified variables in c. Finding a suitable F and proving the associated entailments can be tricky in the abstract but in practice is guided by using a "ramification library" as given in §**??**.

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

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

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

The proof is later

2.3 The existential ogre

Existential rule by Floyd [Floyd 1967] (1967!)

EXISTENTIAL EXTRACTION

$$\frac{\forall x.\, (\{P\}\, c\, \{Q\})}{\{\exists x. P\}\, c\, \{\exists x.\, Q\}}$$

```
// {P}
   c
// {∃x. Q}
// {Q}
```

But:

```
// {G₁}
// ↘ {L₁}
       c
// ↙ {∃x. L₂}
// {∃x. G₂}
// {G₂}
```

RAMIFY-Q

$$\frac{\{L\}\, c\, \{\exists x.\, L'\} \qquad G \vdash L * \big(\forall x.\, (L' -\!* G')\big)}{\{G\}\, c\, \{\exists x.\, G'\}}$$

2.4 Proofs

3. A framework for graph theory

As pointed out in [Hobor and Villard 2013], naïave attempts to verify graph-manipulating programms using the shape-only predicates are unsound. An obvious solution—especially for verifying functional correctness—is to involve a mathematical graph γ in spatial predicates as a parameter. Since the additional parameter is involved in the specification, we needs a way to reason about it, so as to deduce the specification. Thus we formalize a proof framework for mathematical graphs and provide a bunch of useful theorems in graph theory to ease the burden in verifying programs. In this section, we will introduce the framework and show how we built them.

3.1 Definitions of graph and other concepts

The core of the graph theory framework are the definitions of graph, graph-related structures and relations. In mathematics, a (directed) graph is an ordered pair (V, E) where V is a set of vertices or nodes and E is a set of edges comprising a source node and a destination node. In our framework, we defined a similar structure with minor amendment: we attached "validity" property to each vertex and edge while V and E serve as types instead of sets of vertices and edges respectively. There are two reasons for the amendment. The first is that in many proof assistants, it is much easier and natural to declare types instead of sets. When using types, validity is an effective way in controlling membership: most graphs do not contain all instances of vertex type and edge type. The second reason is expanding the scope of representation, not just normal graphs. For example, in some cases, we need to describe the difference of two graphs: $\gamma_1 - \gamma_2$, which is not necessarily a graph because it may contain dangling edges. The "subtracted" nodes are not really removed but are ruled out from valid nodes because they are still referred in edges part of the definition of $\gamma_1 - \gamma_2$. So we call this structure **PreGraph**, which means it may be incomplete in comparison with a classical graph. In our framework, a PreGraph γ is a hextuple $(V, E, \Phi_V, \Phi_E, s, t)$ where Φ_V and Φ_E are validity predicates for vertex type V and edge type E, s and t are functions which takes an edge as argument and returns source/destination node of the edge. Many concepts such as reachability, subgraph and structural equivalence are defined based on PreGraph.

Path Once we have the definition of PreGraph, it is time to define another infrastructure: path. Informally, a path is a list of nodes concatenated with edges. Since an edge contains the information about its source and destination nodes, there is no necessary to define path as an interleaving list of nodes and edges. Then we have two obvious choices for representation of a path: a list of nodes or a list of edges.

Both choices have certain defects. If a path is defined as a list of nodes, we can not distinguish different paths between two nodes in case there are multiple edges between two nodes. If a path is defined as a list of edges, then we can deal with multiple edges but we can not represent an empty path for a certain node—we can not determine which node an empty list of edges belongs to.

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) $-*$ is the adjunct of $*$

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 $(P * Q \vdash R) \Longleftrightarrow (P \vdash Q -* R)$ (2) reduction using modal axioms K and N

To avoid the defects above, we define the path as a ordered pair (n, l) where n is a node and l is a list of edges. A valid path requires that n is the source node of the first edge of l and l is well chained—the destination of an edge in l is the same as the source of the next edge. The list l can be null to represent an empty path for a particular node n.

Why we insist that a path—even an empty path—must have a leading node? One reason is that with such a definition, we can give a consistent definition for a very important concept: reachability.

Reachability Among various properties derived from PreGraph, the most important one is the reachability. The definition of reachability is based on path. The notation $\gamma \models L_{n_2}^{n_1}(P)$ means in PreGraph γ, node n_2 is reachable from node n_1 along the path L while every node in L satisfies predicate P. This notation, along with other derived ones, such as

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

form the bedrock of nearly every nontrivial predicate about graph. Either the relation of two states—before and after running an algorithm—of a graph or the description of a graph with a particular shape, reachability is inevitable. To some extent, it is quite natural because most graph-related algorithms (DFS, BFS, Shortest Path, Spanning Tree, etc) depend on a small operation: exploring neighbours from one node. Thus when describing the effect of an algorithm, reachability is indispensable. Some of those descriptions are discussed in later sections.

Subgraph When we tie a mathematical graph γ to a spatial graph predicate $g(x, \gamma)$ (which will be explained later), g "owns" only the spatial representation of the portion of γ that is reachable from x; γ may contain other nodes. When we reason about $g(x, \gamma)$, it is a very natural requirement to describe the reachable portion of γ in pure part. We generalize this description as two concepts: partial graph $\gamma \uparrow P$ and subgraph $\gamma \downarrow P$ for arbitrary predicate P. To define $\gamma \uparrow P$ and $\gamma \downarrow P$, for any $\gamma = (V, E, \Phi_V, \Phi_E, s, t)$, we do not change V and E but change Φ_V and Φ_E by adding proposition about satisfying P for nodes. It means valid nodes in $\gamma \uparrow P$ and $\gamma \downarrow P$ must satisfy P. The only difference between $\gamma \uparrow P$ and $\gamma \downarrow P$ is that in $\gamma \uparrow P$ the edges with its source node satisfying P is valid, but in $\gamma \downarrow P$, valid edges means both its source and destination nodes satisfy P. With the concepts of partial graph and subgraph, we can express the reachable portion by instantiating P as reachable predicate.

Structural Equivalence Many graph-manipulating algorithms adopt divide and conquer paradigm. Most of those algorithms contain certain invariants as parts of their specifications. Usually it means certain portion of graph before program execution is equivalent to the one after execution. To describe this relation, we introduced "structural equivalence" in our framework with the following definition:

$$\begin{aligned}\gamma_1 \cong \gamma_2 \triangleq & \forall v, \Phi_{V1}(v) \leftrightarrow \Phi_{V2}(v) \wedge \\ & \forall e, \Phi_{E1}(e) \leftrightarrow \Phi_{E2}(e) \wedge \\ & \forall e, \Phi_{E1}(e) \to \Phi_{E2}(e) \to \mathsf{src}_{\gamma_1}(e) = \mathsf{src}_{\gamma_2}(e) \wedge \\ & \forall e, \Phi_{E1}(e) \to \Phi_{E2}(e) \to \mathsf{dst}_{\gamma_1}(e) = \mathsf{dst}_{\gamma_2}(e)\end{aligned}$$

Informally it means γ_1 and γ_2 has the same vertex set and edge set. And any valid edge in both graphs is comprised by the same source and destination nodes. This relation can be used with subgraph and reachability to define many concrete relations in program verification.

3.2 Classification of various graphs

PreGraph and its derived properties (reachability, subgraph, etc) are inadequate for real program verifications. Admittedly many helpful lemmas can be inferred directly from PreGraph. But when we dealing with concrete graphs for various algorithms, there are many features which a bare PreGraph can not inculde. For example, when we compute some properties of a graph, we always hope the graph is finite connected. In the recursive defnition of a graph data structure, a node contains many pointers to point to its neighbors. The pointers could be null to indicate that they do not point to any. Thus we need a special node which represents null pointer. In some cases, we have to specify a graph in which the outdegree of each nodes is 2. All these additional properties are abstracted as different property bundles. We defined **LocalFiniteGraph**, **MathGraph** and **BiGraph** for the three requirements above, respectively.

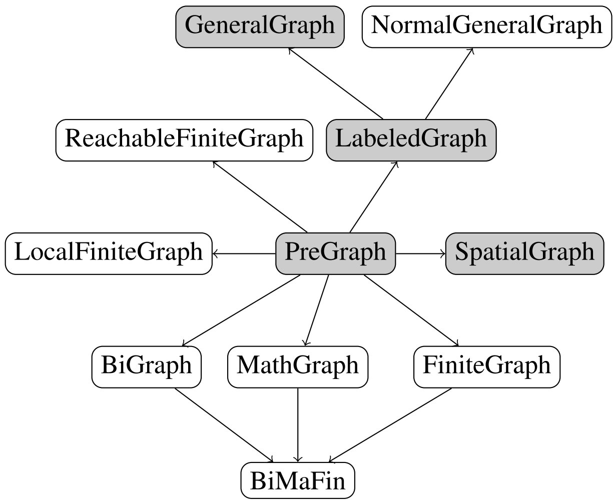

Figure 2. Various Kinds of Graphs

As shown in Figure 2, there are several different "Graph" definitions in our framework. Some of them (with gray background) are real mathematical objects, just like PreGraph. The rest are just property bundles which are PreGraph with special properties. They provide different views of graphs from a certain perspective.

In our framework, there are four kinds of *real* graphs: **PreGraph**, **SpatialGraph**, **LabeledGraph** and **GeneralGraph**. The definition of PreGraph is already known. Based on PreGraph, each node and each edge in a LabeledGraph has extra labels which can be seen as label functions of nodes and edges. A GeneralGraph is a LabeledGraph with a customized sound condition. A SpatialGraph is the graph directly stored in memory, which contains connecting and user specified informations stored on every vertex and/or every edge.

The rest in Figure 2 are all property bundles where LocalFiniteGraph, MathGraph and BiGraph are already explained. A FiniteGraph is a graph with finite number of vertices and edges. (So a FiniteGraph is definitely a LocalFiniteGraph.) A BiMaFin is a graph satisfying BiGraph, MathGraph and FiniteGraph simultaneously. A ReachableFiniteGraph is a graph in which all reachable nodes of an arbitrary node are finite. A NormalGeneralGraph is a special GeneralGraph used for certain algorithms. Usually in the correctness proofs of concrete algorithms, all graphs satisfy the definition of BiMaFin, LocalFiniteGraph, ReachableFiniteGraph and etc. But for many math theorems in graph theory, some properties (e.g. FiniteGraph, BiGraph) are not necessary. These different properties for different theorems make our framework more flexible and general.

3.3 Computable Reachability

We proved quite many property-bundle specific theorems. One typical example of such theorems is the following one:

THEOREM 1. *For any graph γ which is both MathGraph and LocalFiniteGraph and any node x in γ, if the number of nodes reachable from node x is finite, then we can find a set which exactly contains the reachable nodes from x in γ.*

It sounds so trivial but the proof is totally non-trivial. The most obvious way to prove it—filtering reachable nodes from all valid nodes—is impossible: there is no decision procedure for reachability to select reachable nodes because the current definition of graphs is applicable for graphs containing infinite number of nodes (A LocalFiniteGraph can still be infinite). In such a graph, one may need to inspect infinite number of paths to judge whether a node is reachable, which is a mission impossible.

Another intuitive way to construct the reachable list is using a breadth-first searching algorithm to collect distinct nodes along the edges from x, which is employed in the proof. But still, a naïve implementation may not terminate. As mentioned in the premises of this lemma, the number of reachable nodes has an upper bound. So the actually working breadth-first searching function is constructed as follows. It holds a set of nodes collected so far and a queue for visited but unexpanded nodes. It keeps on expanding nodes from the queue to add distinct nodes to result set and the processing queue, until the processing queue is empty or the number of nodes collected reaches the upper bound. It is worth mentioning that constructing this function in Coq is hard. The direct implementation is rejected because it violates the syntactic criteria for recursion in Coq. It has to be defined in a sophisticated way: defining a well-founded relation and proving that the definition of the function fulfills the relation.

After the establishing of the searching function, it is still hard to prove that the result generated by the function is the reachable list. There are two goals that needs to be proved: one is that all nodes collected by this searching function is reachable from x and the other is all reachable nodes from x is in the collected list of the searching function. The former one is relatively easier than the latter because for the latter case, there are two completeness proofs corresponding to the two different termination conditions. It is not known yet whether a proof by contradiction exists or is simpler.

3.4 Application of the framework

Graph-manipulating programms may also deal with other structures, such as dag (directed acyclic graph), tree or spanning tree. With the basic defintions in our framework, new structures can be defined easily. Acyclic graph is just a graph with an additional property: forall any x and y, if x is reachable from y, then y is not reachable from x. Tree is similar. The additional property for tree is that there is one and only one path from root to any reachable node from root. Spanning tree is a tree with the same reachable vertex set of the original graph.

But when dealing with formal proofs, there are some unexpected facts. For example, our naïve definition of the predicate `spanning_tree` is not strong enough to complete the inference. We express that graph g_2 is a spanning tree of graph g_1 starting from root node r as `spanning_tree` $g_1\, r\, g_2$. We conclude three relations as the predicate. First, the unreachable parts from root r in g_1 and g_2 are the same. Secondly, the shape of g_2 must be a tree. Lastly, all nodes reachable from r in g_1 are still reachable from r in g_2. The second relation ensures that the result is a tree while the third one ensures it is a spanning tree of g_1. At first glance, it is a very complete definition. But the subsequent formal proof reveals that the definition still lacks one condition: for any two nodes a and b, if a is reachable from r in g_1 and b is not reachable from r in g_1, then in g_2, b is not reachable from a. It is a long-winded but necessary relation.

4. Defining and reasoning about spatial graphs

To prove the functional correctness of real graph-manipulating algorithms, we provide spatial predicate of graphs as a shape description about heaps. As a matter of fact, we defined a much more general spatial predicate "★" to indicate a collection of standard points-to predicates chained by $*$ in separation logic. The spatial graph predicate is just a special case in terms of ★.

4.1 Traditional fixpoints fail

Hobor and Villard[Hobor and Villard 2013] defined the separation logic graph predicate $\mathtt{graph}(x, \gamma)$ in direct analogy to the standard separation logic definition of a tree as follows:

$$\mathtt{graph}(x,\gamma) \triangleq (x = 0 \wedge \mathtt{emp}) \vee \exists d,l,r.\gamma(x) = (d,l,r) \wedge \\ x \mapsto d,l,r \circledast \mathtt{graph}(l,\gamma) \circledast \mathtt{graph}(l,\gamma)$$

where γ is a mathematical graph and $\gamma(x)$ extracts the data mapped by the label function and two neighbors of node x. However, it is peculiarly challenging in rigorously formalizing `graph` as shown above.

Recursive/inductive 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, 95% 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 ("$\triangleq$").

Appel and McAllester developed an additional fixpoint μ_{R} [Appel and McAllester 2001] whose [Appel et al. 2007] mechanically verified its soundness. People can still define recursive predicate P through F_p and μ_{R}, but this time the F_p needs to be *contractive*. Informally, a contractive function is one such that if τ is approximately equal to σ, then $F_p(\tau)$ is more accurately equal to $F_p(\sigma)$. The approximate equality is achieved by a data type as a sequence of accurate approximations taken successively. This idea is called step-indexing.

We attempted to formulate `graph` through fixed-point functions μ_{T} and μ_{R}. The contractive functor `graphF` is defined as follows:

$$\mathtt{graphF}(Q,x,\gamma) \triangleq (x = 0 \wedge \mathtt{emp}) \vee \\ \exists d,l,r.\gamma(x) = (d,l,r) \wedge x \mapsto d,l,r \circledast \rhd Q(l,\gamma) \circledast \rhd Q(r,\gamma)$$

where $\rhd$ is is the "later" operator which implements the machinery of step-indexing. Note that `graphF` is a normal predicate without recursion. `graph` is defined as $\mu_{\mathsf{R}}\, \mathtt{graphF}$. One advantage of this definition of `graph` is that proof by induction is possible because the step-index can be seen as the inductive number. Unfortunately `graph` is not *precise* under this definition. For any spatial predicate P, $\mathrm{precise}(P)$ means whenever P is satisfied on a sub-state, that sub-state must be unique. Being precise is a crucial requirement of `graph` for key theorems in our framework. Further-more, it can be proved that for any predicate P, $\rhd P$ is not precise. So this defintion is abandoned.

Similarly we can define a covariant functor `graphQ` as follows:

$$\mathtt{graphQ}(Q,x,\gamma) \triangleq (x = 0 \wedge \mathtt{emp}) \vee \\ \exists d,l,r.\gamma(x) = (d,l,r) \wedge x \mapsto d,l,r \circledast Q(l,\gamma) \circledast Q(r,\gamma)$$

The only difference between `graphQ` and `graphF` is that `graphQ` does not have the $\rhd$ operator. With this definition `graph` can be defined as $\mu_{\mathsf{T}}\, \mathtt{graphQ}$. Again we need to prove the preciseness of `graph`. Since there is no induction principle for this definition, we tried to prove it through the following lemma:

$$\mathtt{graph}(x,\gamma) \dashv\vdash \bigstar_{v \in \mathit{reach}(\gamma,x)} v \mapsto \gamma(v) \tag{2}$$

where $\mathit{reach}(\gamma, x)$ is the set of nodes reachable from x in γ and the definition of ★ over a set and a predicate p is

$$\bigstar_{\{a_1,a_2,\ldots,a_n\}} p \triangleq p(a_1) * p(a_2) * \ldots * p(a_n).$$

The preciseness of `graph` is a natural corollary of the lemma above because for any v, $v \mapsto \gamma(v)$ is precise. Unfortunately that lemma does not hold even for a self-referencing single node graph because: every time the expanding of $\mathtt{graph}(x, \gamma)$ leads to itself. So this definition is abandoned too.

4.2 The iterated separating conjunction

The two failures of fixpoint method above force us to turn to another direction. Inspired by lemma 2, we defined the `graph` as follows:

$$\mathtt{graph}(x, \gamma) \triangleq \mathop{\bigstar}_{v \in reach(\gamma, x)} v \mapsto \gamma(v)$$

This non-recursive predicate says that a graph whose root is x is a list of reachable nodes from x separated by $\ast$. From this definition we can prove the unfold lemma:

$$\begin{array}{l}\mathtt{graph}(x, \gamma) \Leftrightarrow \exists d, l, r. \gamma(x) = (d, l, r) \wedge \\ x \mapsto d, l, r \mathbin{\cup\!\!\!*} \mathtt{graph}(l, \gamma) \mathbin{\cup\!\!\!*} \mathtt{graph}(r, \gamma)\end{array}$$

1.2. `Iter_sepcon` and `pred_sepcon` are defined. And related ramification rules are proved. 1.3. The most general graph-spatial-predicate `vertices_at` are defined (for all possible styles of graphs). Related ramification rules are proved. Graph and graphs are defined as special cases of vertices at.

2. A minor implementation trick. There are many tactics defined in `msl_ext/ramify_tactics.v`, which can manipulate low level heaps efficiently.

* Separating the material into the general vs. tool-specific part. Measurements of etc.

5. Ramification Rules

$$\textsc{Frame}\ \dfrac{\begin{array}{c}\{P\}c\{Q\}\\ F \text{ is stable w.r.t. } \mathsf{ModVar}(c)\end{array}}{\{P * F\}c\{Q * F\}}$$

$$\textsc{Ramification}\ \dfrac{\begin{array}{c}\{L\}c\{L'\}\\ G \vdash L * (L' - * G')\\ (L' - * G') \text{ is stable w.r.t. } \mathsf{ModVar}(c)\end{array}}{\{G\}c\{G'\}}$$

$$\textsc{Ramification-P}\ \dfrac{\begin{array}{c}\{L\}c\{L'\}\\ G \vdash L * \Box^{[\![c]\!]}(L' - * G')\end{array}}{\{G\}c\{G'\}}$$

5.1 P for Pure Facts

Separation logic has been mechanized by many projects CITE CITE CITE. In many of them, like VST and Charge!, expressing the value of a local variable (a variable stored in stack) is a pure fact rather than a spatial fact. Because the side condition of ramification rule requires $(L' - * G')$ to be stable w.r.t. modified local variables in c[1], it is almost impossible to apply ramification rule in any practical situations in these systems. In this paper, we present a pure-facts-related rule (we call it ramification-P rule, or just P rule, in the rest of this paper) such that it is sound and practical in the most general setting of separation logics.

The primary ramification rule is essentially an application of the frame rule using $(L' - * G')$ as frame. Thus, the key point of handling pure facts is to find a legal frame even if $(L' - * G')$ is not stable w.r.t. $\mathsf{ModVar}(c)$. This frame is $\Box^{[\![c]\!]}(L' - * G')$ in ramification-P rule.

$$\begin{array}{rcl} m \models \Box^R P & \Leftrightarrow & \forall m',\ \text{if } m \xrightarrow{R} m' \text{ then } m' \models P \\ m \xrightarrow{[\![c]\!]} m' & \Leftrightarrow & m \text{ and } m' \text{ coincide everywhere} \\ & & \text{except } \mathsf{ModVar}(c) \\ P \text{ is stable} & \Leftrightarrow & \forall m\ m', \text{if } m \text{ and } m' \text{ coincide everywhere} \\ \text{w.r.t. } S & & \text{except } S, \text{ then } m \models P \text{ iff } m' \models P \end{array}$$

Here, $\Box$ represents the necessity modal operator. The formula $\Box^{[\![c]\!]}(L' - * G')$ says, it is true on a state m if and only if for any state m', if m and m' coincide everywhere except on the variables modified by c, then $(L' - * G')$ is true on m'.

Based on the combination frame rule, consequence rule and three basic facts below, we can immediate prove ramification-P rule.

(a) $\Box^{[\![c]\!]}(L' - * G')$ is stable w.r.t. $\mathsf{ModVar}(c)$.[2]
(b) $G \vdash L * \Box^{[\![c]\!]}(L' - * G')$. (Assumption)
(c) $L' * \Box^{[\![c]\!]}(L' - * G') \vdash G'$. [3]

5.2 Establish the Assumption Entailment of P Rule

It is well-known that the proof theory with magic wand is already complicated, so generally speaking, it will not be a easy task to prove an entailment with magic wand together with modality. However, people need to prove an entailment with form

$$G \vdash L * \Box^R(L' - * G') \tag{3}$$

at first when applying ramification-P rule. Luckily, this special form makes the task simpler.

First of all, SOLVE-RAM-P rule can turn the proof goal into two wand-free and modality-free entailments. Specifically, people only need to find an R-stable predicate F, such that $G \vdash L * F$ and $F * L' \vdash G'$ are both true.

[1] In previous papers, the side conditions of Frame rule and ramification rule are usually expressed as "$\mathsf{FreeVar}(F) \cap \mathsf{ModVar}(c) = \varnothing$" and "$\mathsf{FreeVar}(L' - * G') \cap \mathsf{ModVar}(c) = \varnothing$". The side conditions used in this paper are equivalent with typical ones if the semantic interpretation of $\mathsf{FreeVar}$ is used. All the previous mentioned projects takes semantic interpretation instead of syntactical interpretation.

[2] This can be proved directly from the definition of $[\![c]\!]$ and stability, and the fact that $[\![c]\!]$ is an equivalence relation.

[3] When R is reflexive, T-Axiom of modal logic is sound, i.e. for any P, $\Box^R P \vdash P$. As $[\![c]\!]$ is reflexive, we know the fact that $\Box^{[\![c]\!]}(L' - * G') \vdash L' - * G'$, which is immediate followed by $L' * \Box^{[\![c]\!](L' - * G')} \vdash G'$.

SOLVE-RAM-P alone is not a satisfactory proof theory because in that case using P rule would have no different from using frame rule directly. The key point here is that, an entailment with form **??** can be proved in a modularized way. For primary ramification rule, CITE proposed two proof rule, RAM-FRAME and RAM-SPLIT[4], to divide an entailment with form $G \vdash L * (L' - *G')$ into small pieces. When it comes to ramification-P rule, two corresponding proof rules, RAM-P-FRAME and RAM-P-SPLIT are still sound.

$$\text{SOLVE-RAM-P}\ \frac{\begin{array}{c} G \vdash L * F \\ F * L' \vdash G' \\ F \text{ is stable w.r.t. } \mathsf{ModVar}(c) \end{array}}{G \vdash L * \square^{[\![c]\!]}(L' - *G')}$$

$$\text{RAM-P-FRAME}\ \frac{\begin{array}{c} G \vdash L * \square^{[\![c]\!]}(L' - *G') \\ F \text{ is stable w.r.t. } \mathsf{ModVar}(c) \end{array}}{G * F \vdash L * \square^{[\![c]\!]}(L' - *G' * F)}$$

$$\text{RAM-P-SPLIT}\ \frac{\begin{array}{c} G_1 \vdash L_1 * \square^{[\![c]\!]}(L'_1 - *G'_1) \\ G_2 \vdash L_2 * \square^{[\![c]\!]}(L'_2 - *G'_2) \end{array}}{G_1 * G_2 \vdash L_1 * L_2 * \square^{[\![c]\!]}(L'_1 * L'_2 - *G'_1 * G'_2)}$$

To conclude, if L' and G' are two separating conjunctions of a bunch of atomic predicates, RAM-P-FRAME and RAM-P-SPLIT can establish **??** from entailments with the same form but smaller size. Atomic sized entailments can be proved using SOLVE-RAM-P. They are usually general purposed entailments and do not need to be proved for every single program. In section 6, we will see examples of this approach for real programs.

5.3 Q for Quantifiers

In secion ???, we have already seen that it is a practical approach writing pre/postconditions as a separating conjunction of a list of atomic predicates (which makes RAM-P-FRAME and RAM-P-SPLIT useful). But unfortunately, an existential in post condition (also very common as we have seen in section ???) will prevent us from using these two rules. Now, one natural solution is to find other proof rules, like the following one, to deal with existential quantifiers.

UNSOUND-RAM-Q-SPLIT

$$\frac{\begin{array}{c} G_1 \vdash L_1 * (\exists x, L'_1(x) - *\exists x, G'_1(x)) \\ G_2 \vdash L_2 * (\exists x, L'_2(x) - *\exists x, G'_2(x)) \end{array}}{G_1 * G_2 \vdash L_1 * L_2 * (\exists x, L'_1(x) * L'_2(x) - *\exists x, G'_1(x) * G'_2(x))}$$

But this rule is NOT sound (even though we have not add $\square$ operator to deal with local variable related stuff). The reason is that, given the local piece of memory satisfies $L'_1(x) * L'_2(x)$ for some specific x, we know that it can be split into two small piece of memory and they satisfies $L'_1(x)$ and $L'_2(x)$ respectively. Then the assumption tell us that the global piece can be split into two corresponding piece, $G'_1(x_1)$ and $G'_2(x_2)$ are true on them for some specific x_1 and x_2. Now the problem comes. Only if we could prove $x_1 = x_2$, we could prove the conclusion. But we cannot.

The key point of the failure above is that the frame, $\exists x, L'(x) - *\exists x, G'(x)$, says if $L'(x)$ is true on local then there is another (might be same one) x_0 such that $G'(x_0)$ is true on global. This is too weak for modularity. In many practical cases, we can in fact prove that $G'(x)$ should be true for the exact same x. This observation brings us to the ramification-PQ rule here.

$$\textsc{Ramification-PQ}\ \frac{\begin{array}{c} \{L\}c\{\exists x, L'(x)\} \\ G \vdash L * \square^{[\![c]\!]}(\forall x, L'(x) - *G'(x)) \end{array}}{\{G\}c\{\exists x, G'(x)\}}$$

PQ rule can be directly derived from P rule by using the following theorem from separation logic[5].

$$\forall x, (L'(x) - *G'(x)) \vdash \exists x, L'(x) - *\exists x, G'(x)$$

Like what we do to P rule, three corresponding rules, SOLVE-RAM-PQ, RAM-PQ-FRAME and RAM-PQ-SPLIT, are proved sound and can be used to establish the assumption of PQ rule in a modularized way. For those who do not care about local variable related issue, a ramification-Q rule can be used to deal with existentials. For the sake for space here, we omit them in this paper.

5.4 Ramification in Decorated Programs

One nice thing about Hoare logic is that it enables people to write combinational proofs. Moreover, such kind of proofs can be written in a nice printed form, decorated programs.

By adding a new pattern, we call it localized and unlocalize, ramification proofs can also be presented in a decorated programs.

Figure 3 shows such a decorated program. We call the action in line 2 *localize* and call the action in line 6 *unlocalize*. A Hoare logic proof using ramification rule can always be written as a decorated program with localize and unlocalize, as long as wherever we write do unlocalize action, we should prove a side condition, e.g. $G_1 \vdash L_1 * (L_2 - *G_2)$ in this example.

[4]

$$\text{RAM-FRAME}\ \frac{\begin{array}{c} G \vdash L * (L' - *G') \\ F \text{ is stable w.r.t. } \mathsf{ModVar}(c) \end{array}}{G * F \vdash L * (L' - *G' * F)}$$

$$\text{RAM-SPLIT}\ \frac{\begin{array}{c} G_1 \vdash L_1 * (L'_1 - *G'_1) \\ G_2 \vdash L_2 * (L'_2 - *G'_2) \end{array}}{G_1 * G_2 \vdash L_1 * L_2 * (L'_1 * L'_2 - *G'_1 * G'_2)}$$

[5]

$$\frac{\dfrac{\dfrac{\forall x, (L'(x) - *G'(x)) \vdash L'(x_0) - *G'(x_0)}{\forall x, (L'(x) - *G'(x)) * L'(x_0) \vdash G'(x_0)}}{\forall x, (L'(x) - *G'(x)) * \exists x, L'(x) \vdash \exists x, G'(x)}}{\forall x, (L'(x) - *G'(x)) \vdash \exists x, L'(x) - *\exists x, G'(x)}$$

```
{    $G_1$    }
↘ {    $L_1$    }
        c1;
        ...
        c5;
↙ {    $L_2$    }
{    $G_2$    }
```

$$\frac{\begin{array}{c}\{L_1\}\,c1;...;c5\,\{L_2\}\\ G_1 \vdash L_1 * (L_2 -*G_2)\end{array}}{\{G_1\}\,c1;...;c5\,\{G_2\}}$$

Figure 3. Localize and unlocalize in decorated programs

6. Ramification based on VST

6.1 Background: Verified Software Toolchain

VST is a correctness-certified tool to prove functional correctness of C programs CITE. All Hoare rules are proved sound and users can use them to build modularized proof. At the same time, users can have all the convenience offered by separation logic. For example, frame rule is already proved sound as well. VST is fully developed in Coq and it uses the C semantics offered by ComCert. CITE

Apart from enabling mechanized program verification in Coq, VST establishes a connection between Hoare logic proofs and decorated programs. Specifically, when users prove a Hoare triple, VST's tactic system enables them to feel as if they were write a decorated program from up to down, but the proof built in Coq has a structure as an inference diagram.

For example, when the proof goal is $\{P_1\}\,c1; c2\,\{P_5\}$, VST's user can apply some Hoare rule to get a triple for c1, e.g. $\{P_1\}\,c1\,\{P_2\}$. VST's tactic system then applies sequence rule automatically and the proof goal left to user will be $\{P_2\}\,c2\,\{P_5\}$. On user's view, his/her proof goal changes from $\{P_1\}\,c1; c2\,\{P_5\}$ to $\{P_2\}\,c2\,\{P_5\}$ and these interaction with VST system is exact the same as writing the first three lines of his/her decorate program on a pen-and-paper proof. At the same time, in Coq's underlying logic, VST's tactic system builds a proof tree from bottom to the top.

In summary, VST's users build a Hoare logic proof by interacting with VST's tactic system. At any intermedium point of this interaction process, the decorated program is partially done (from top to bottom) and the inference tree is also partially done (the holes are proof goals in Coq).

6.2 Extend VST to Support Ramification

In order to extend VST to support ramification, we should enable people to write decorated programs with localize and unlocalize action. Our task here is to construct a proof in Coq's underlying logic from a decorated program, in which localize and unlocalize are involved. Moreover, our extension of VST's tactic system should construct a partial proof when the user finishes part of his proof.

It is especially difficult when the following kind of partial decorated programs are considered.

```
{    $P_1$    }
     c1
{    $P_2$    }
↘ {    $P_3$    }
       c2;
       {    $P_4$    }

       ...

   (to be done: c3; c4; c5)

{    Post    }
```

Our tactic system cannot even know when corresponding unlocalize action will be done. To construct a partial proof, the tactic system cannot know where to close this ramification block.

In order to solve this problem, our tactic systems builds the partial proof in underlying logic by using uninstantiated frame. For example, the partial decorated program above is treated like this:

```
{    $P_1$    }
     c1
{    $P_2$    }
{    $?F * P_3$    }
     c2;
{    $?F * P_4$    }

     ...

   (to be done: c3; c4; c5)

{    Post    }
```

2.4. VST instance of `pSpatialGraph_Graph_Bi` and `sSpatialGraph_Graph_Bi` are constructed in `"spatial_graph_aligned_bi_VST.v"` and `"spatial_graph_`

3. Embed ramification into VST. 3.1. Ramification rule are proved sound in VST. 3.2. A special ramification rule for VST's Sep-Local-Prop style pre/post condition is prove. The point is traditional ramification rule require the whole frame-like-wand-expression to be closed w.r.t. the modified variables. This special rule split closed and unclosed away. 3.3. Localize and unlocalize are defined. 3.3.1. Localize/unlocalize offer a user-friendly way of using ramification rule. 3.3.2. Unlocalize tactic need "Grab Existential Variables" afterwards. It is not nice. 3.3.3. Writing Ocaml plugin is one solution. But we need to develop for both mac and windows. 3.3.4. Or we can see whether Coq's next version offers more tactics for existential variables.

7. Enabling externally-verified lemmas in HIP/SLEEK

* the connection to HIP/SLEEK

In the H/S section we talk about the engineering inside H/S, the module type/module interface, forward ramify, etc.

8. Applying ramification

5. Mark algorithm 5.1. For Ramification-Paper-style proof 5.1.1. Math land theorems for marking algorithm (general situation and bi-graph situation) are all proved. Mainly in "marked_graph.v" and "spatial_graph_mark_bi.v". 5.1.2. Ramification rule for marking algorithm (bi-graph situation) are all proved in "spatial_graph_mark_bi.v". 5.1.3. Combining 2.4 and 3.1.1 and 3.1.2, we have a end-to-end proof for marking-graph in VST. 5.1.4. We have an end-to-end proof for marking-dag, but not defining dag predicate as a whole. 5.1.5. The module type which will be generated by HIP/SLEEK should be instantiated by 2.2 and 2.5.

6. Spanning tree algorithm 6.1. We divide the spanning tree relation into structural part and marking part. They are both defined properly. 6.2. Important pure facts and ramification rules are not proved yet. 6.3. Shengyi has already known how to use VST to handle the C program of bigraph spanning tree.

9. Related and future work

The most famous graph related theorem which has been mechanically verified is the Four Color Theorem: Any planar map can be colored with only four colours. In 2005, Benjamin Werner and Georges Gonthier formalized a proof of the theorem [Gonthier 2005] inside Coq. It is very easy and natural to rephrase the problem in graph theory: by taking regions as nodes and connecting each pair of adjacent regions as edges, coloring the map is equivalent to coloring the graph obtained. However, they used a different kind of combinatorial structure, known as hypermaps, instead of graphs. Basically, a hypermap is a type "dart" with several functions mapping dart to dart. The combinatorial and geometrical properties are encoded as certain permutation properties of those functions. It is quite a very different structure from graph.

Lars Noschinski built a formalized graph library for the Isabelle/HOL proof assistant and verified a method of checking Kuratowski subgraphs used in the LEDA library. It supports general infinite directed graphs with labeled and parallel arcs [Noschinski 2015a]. His definition of graph is similar to our PreGraph except he uses vertex/edge set instead of validity functions. Besides, Noschinski's library also covers basic graph related concepts such as reachable component and spanning tree.

Nordhoff and Lammich [Nordhoff and Lammich 2012] formalized and proved Dijkstra's algorithm in Isabelle. Their graph is defined as vertex and edge sets where the edge is a triple (source, label, destination). They only defined what they need for the algorithm.

Written in HOL, Wong [Wong 1991] expressed a small portion of the conventional graph theory, which is mainly used to model the railway track network and applied in signalling systems. It does not contain too many graph property-related theorems.

Chou [Chou 1994] formalized theory of undirected graphs in HOL that emphasize on the notion and important properties of trees. He applied this library to verified distributed algorithms [Chou 1995].

Duprat [Duprat 2001] formalized graph in an inductive way in Coq. Only some basic properties are proved in it. To our knowledge, no application is built on it.

In [Yamamoto et al. 1995] a formalization of planar graph is inductively defined in HOL. They use it to prove Euler's formula as an application. [Tamai 2000] treat the same problem. But their purpose is just giving a formal specification in CafeOBJ. So their graph library only contains formal definitions.

In [Yamamoto et al. 1998], Yamamoto et al formalized directed graph based on Wong's work [Wong 1991]. They proved the correctness of the abstract A* algorithm based on graph and semi-lattice. [Tamai 2000]

NASA's graph theory library is written in PVS [R. W. and J. A. 1998]. It is restricted to finite graphs only and does not support multi-edge graphs. They use the library to prove Ramsey's Theorem and Menger's Theorem.

[Bauer and Nipkow 2002] inherited the inductive approach of [Yamamoto et al. 1995] for the formalization of planar graph theory. They formally proved the 5 color theorem using graph theory and triangulations.

In [Nipkow et al. 2006], a finite, undirected, planar graph is formalized as a list of faces and faces as lists of vertices. That library is mainly used to prove the completeness of the enumeration of tame graphs.

[Ridge 2005] also mechanised graphs and trees in Isabelle/HOL. It is closed to [Wong 1991] with the following difference: the edges are represented as sets of vertices instead of atomic objects.

In [Noschinski 2015b], Noschinski presented a graph library in Isabelle/HOL to reason about graphs and implemented a verified decision procedure for combinatorial planarity of graphs. He also verified checkers for both the planarity and the non-planarity certificates emitted by the LEDA library. Both the implementation and verification of the checker are written in the abstract language of AutoCorres.

In [Dubois et al. 2015] they proposed a formalization of graphs without multiple edges. A formally verified auditor was developed to certify the result of a function that calculates a maximum cardinality matching. The executable code of the auditor is extracted from Coq directly.

5.2. An alternative way of verifying marking program is reasoning about the whole history of marking operations. The disadvantage of it is that it currently needs more work in a Hoare logic framework. The advantage of it is that its reasoning structure are more similar with the way we understand it in our first algorithm class. 5.3. I take some effort on garbage collector like graph structural. Though it is only connecting this special structural with 5.1.1 and 1.3, it takes much time and it is not finished yet.

Local variables. Hobor and Villard hacked their way around this [local var] issue 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 or to do a ramification across multiple assignments as we do in lines 17–21 of figure 1.

An alternative way to avoid local variable issues is to use “variables as resource” [Bornat et al. 2006]. Unfortunately variables as resource introduces other unpleasantness, which is why many mechanized verification systems do not use it[Appel et al. 2014; Bengtson et al. 2012].

Existentials. Hobor and Villard were able to avoid having existentials in localized postconditions because they defined all of their mathematical operations (*e.g.* $mark$, $mark1$) as functions rather than relations.

10. Conclusion

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.

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.

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 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.

C.-T. Chou. A formal theory of undirected graphs in higher-order logic. In *Higher Order Logic Theorem Proving and Its Applications*, pages 144–157. Springer, 1994.

C.-T. Chou. Mechanical verification of distributed algorithms in higher-order logic. *The Computer Journal*, 38(2):152–161, 1995.

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.

J. Duprat. A coq toolkit for graph theory. *Rapport de recherche*, 15, 2001.

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.

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.

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.

T. Nipkow, G. Bauer, and P. Schultz. Flyspeck i: tame graphs. In *International Joint Conference on Automated Reasoning*, pages 21–35. Springer, 2006.

B. Nordhoff and P. Lammich. Dijkstra's shortest path algorithm. *Archive of Formal Proofs*, Jan. 2012. ISSN 2150-914x. http://isa-afp.org/entries/Dijkstra_Shortest_Path.shtml, Formal proof development.

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.

B. R. W. and S. J. A. A pvs graph theory library. Technical report, 1998.

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

T. Ridge. Graphs and trees in isabelle/hol. 2005.

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

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.

M. Yamamoto, K. Takahashi, M. Hagiya, S.-y. Nishizaki, and T. Tamai. Formalization of graph search algorithms and its applications. In *International Conference on Theorem Proving in Higher Order Logics*, pages 479–496. Springer, 1998.

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

A. Junk

Universally-quantified metavariables can appear free in the predicates to make further connections. Assuming that the abstracted pre- and postconditions A, B, C, and D above all use x, we proceed as follows. First we introduce a new fresh metavariable x whose value will be equal to x after the localization, and then choose $F \triangleq [\mathrm{x} \vdash> x](C - *D)$, that is we substitute the program variable x for the metavariable x. Since we have substituted away x, F ignores it and so we satisfy the side condition on Solve Ramify-P. We then must strengthen C into $C' \triangleq C/|\mathrm{x} = x$ to make the connection at the appropriate program point. Now we are left with the entailments

$$\begin{array}{lcl} \mathrm{x} = 5/|A & \vdash & (\mathrm{x} = 5/|B) * F \\ F & \vdash & (\mathrm{x} = 6/|C') - *(x = 6/|D) \end{array}$$

To further relate the earlier and later values of $\mathtt{x}$ in F we can introduce a second fresh x' and use $B' \triangleq B/|\mathtt{x} = x'$.

B. Remaining proof of RAMIFY-PQ

See figure 4.

$$\dfrac{L_1 \vdash L_1 \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{[x \vdash> x_0](L_2 - *G_2) \vdash [x \vdash> x_0](L_2 - *G_2)}{\forall x.\ (L_2 - *G_2) \vdash [x \vdash> x_0](L_2 - *G_2)}\;\forall\mathsf{e}}{\forall x.\ (L_2 - *G_2) \vdash ([x \vdash> x_0]L_2) - *([x \vdash> x_0]G_2)}\;\textsc{substitute}}{\big(\forall x.\ (L_2 - *G_2)\big) * [x \vdash> x_0]L_2 \vdash [x \vdash> x_0]G_2}\;(3)}{\big(\forall x.\ (L_2 - *G_2)\big) * [x \vdash> x_0]L_2 \vdash \exists x.\ G_2}\;\exists\mathsf{i}}{\big(\forall x.\ (L_2 - *G_2)\big) * (\exists x.\ L_2) \vdash \exists x.\ G_2}\;\exists\mathsf{e}}{\forall x.\ (L_2 - *G_2) \vdash (\exists x.\ L_2) - *(\exists x.\ G_2)}\;(3)}{\vdash \big(\forall x.\ (L_2 - *G_2)\big) \Rightarrow \big((\exists x.\ L_2) - *(\exists x.\ G_2)\big)}\;\Rightarrow\mathsf{i}}{\vdash \llbracket c \rrbracket\Big(\big(\forall x.\ (L_2 - *G_2)\big) \Rightarrow \big((\exists x.\ L_2) - *(\exists x.\ G_2)\big)\Big)}\;\mathsf{N}}{\vdash \Big(\llbracket c \rrbracket\big(\forall x.\ (L_2 - *G_2)\big)\Big) \Rightarrow \Big(\llbracket c \rrbracket\big((\exists x.\ L_2) - *(\exists x.\ G_2)\big)\Big)}\;\mathsf{K}}{\llbracket c \rrbracket\big(\forall x.\ (L_2 - *G_2)\big) \vdash \llbracket c \rrbracket\big((\exists x.\ L_2) - *(\exists x.\ G_2)\big)}\;\mathsf{i}\Rightarrow}{L_1 * \llbracket c \rrbracket\big(\forall x.\ (L_2 - *G_2)\big) \vdash L_1 * \llbracket c \rrbracket\big((\exists x.\ L_2) - *(\exists x.\ G_2)\big)}\;*\ \textsc{split}$$

Figure 4. Remaining proof of RAMIFY-PQ