The Ramifications of Mechanizing Verification

Abstract

Blah blah blah

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.

Contributions and structure of the remainder of this paper.

- Example §2. 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.

```
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) { // {graph(x,γ)}
7    struct Node * l, * r; int root_mark;
8    if (x == 0) return;
9  // {graph(x,γ)/|∃m,l,r. γ(x) = (m,l,r)}
10 // {graph(x,γ)/|γ(x) = (m,l,r)}
11 // ↘ {x ⊢> m,l,r}
12       root_mark = x -> m;
13 // ↙ {x ⊢> m,l,r/|m = root_mark}
14 // {graph(x,γ)/|γ(x) = (m,l,r)/|m = root_mark}
15   if (root_mark == 1) return;
16 // {graph(x,γ)/|γ(x) = (0,l,r)}
17 // ↘ {x ⊢> 0,l,r/|γ(x) = (0,l,r)}
18       l = x -> l;
19       r = x -> r;
20       x -> m = 1;
21 // ↙ {x ⊢> 1,l,r/|γ(x) = (0,l,r)/|∃γ'. mark1(γ,x,γ')}
22 // {∃γ'. graph(x,γ')/|γ(x) = (0,l,r)/|mark1(γ,x,γ')}
23 // {graph(x,γ')/|γ(x) = (0,l,r)/|mark1(γ,x,γ')}
24 // ↘ {graph(l,γ')}
25       mark(l);
26 // ↙ {∃γ''. graph(l,γ'')/|mark(γ',l,γ'')}
27 // { ∃γ''. graph(x,γ'')/|γ(x) = (0,l,r)/|
   //   mark1(γ,x,γ')/|mark(γ',l,γ'') }
28 // { graph(x,γ'')/|γ(x) = (0,l,r)/|
   //   mark1(γ,x,γ')/|mark(γ',l,γ'') }
29 // ↘ {graph(r,γ'')}
30       mark(r);
31 // ↙ {∃γ'''. graph(r,γ''')/|mark(γ'',r,γ''')}
32 // { ∃γ'''. graph(x,γ''')/|γ(x) = (0,l,r)/|
   //   mark1(γ,x,γ')/|mark(γ',l,γ'')/|mark(γ'',r,γ''') }
33 } // {∃γ'''. graph(x,γ''')/|mark(γ,x,γ''')}
```

Figure 1. Clight code and proof sketch for bigraph mark. The steps that induce ramifications are indicated with `$_i$, where the associated ramification entailment is equation number i.

- 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 and mechanizing ramification

Mark example. In Qinxiang's new format.

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.

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

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

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:

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

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.

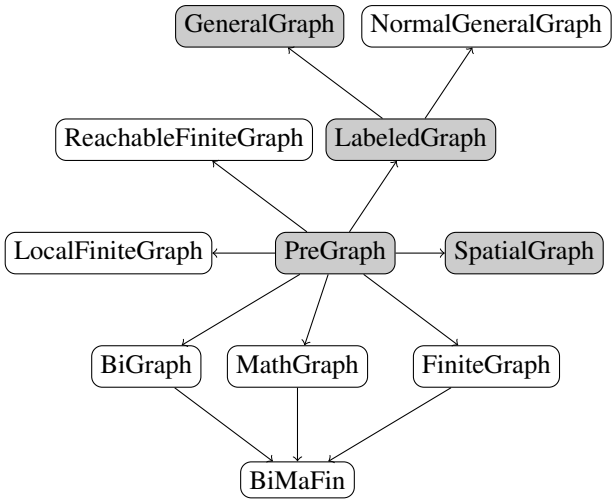


Figure 2. Various Kinds of Graphs

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 $\star$ 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 $\texttt{graph}(x, \gamma)$ in direct analogy to the standard separation logic definition of a tree as follows:

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

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 μ_{T} and μ_{R}. The covariant functor `graphF` is defined as follows:

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

Note that `graphF` is a normal predicate without recursion. The first try is to define `graph` as μ_{T} `graphF`. Unfortunately There is no way to prove any non-trivial theorem even for a self-referencing single node graph because there is no induction principle for this definition: every time the expanding of $\texttt{graph}(x, \gamma)$ leads to itself. So this definition is abandoned.

4.2 The iterated separating conjunction

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. Verifying graph-manipulating programs

4. General Strategy for Verifying Programs 4.1. Using relation. (Better for proof's code reuse). For example, spanning tree is mark together with some structural requirement. 4.2. Using existential quantifier in post condition.

6. Ramification Rules

$$\text{Frame}\ \frac{\{P\}c\{Q\}}{\{P * F\}c\{Q * F\}}\quad F \text{ is stable w.r.t. ModVar}(c)$$

$$\text{Ramification}\ \frac{\{L\}c\{L'\}\quad G \vdash L * (L' -\!\!* G')}{\{G\}c\{G'\}}\quad (L' -\!\!* G') \text{ is stable w.r.t. ModVar}(c)$$

$$\text{Ramification-P}\ \frac{\{L\}c\{L'\}\quad G \vdash L * \Box^{[\![c]\!]}(L' -\!\!* G')}{\{G\}c\{G'\}}$$

6.1 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. ModVar(c). This frame is $\Box^{[\![c]\!]}(L' -\!\!* G')$ in ramification-P rule.

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

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.

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

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

6.2 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

$$G \vdash L * \square^{R}(L' - *G') \tag{1}$$

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.

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.

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

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

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

[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, $\square^R P \vdash P$. As $[\![c]\!]$ is reflexive, we know the fact that $\square^{[\![c]\!]}(L' - *G') \vdash L' - *G'$, which is immediate followed by $L' * \square^{[\![c]\!](L' - *G')} \vdash G'$.

[4]

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

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

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 7, we will see examples of this approach for real programs.

6.3 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.

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

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

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

[5]

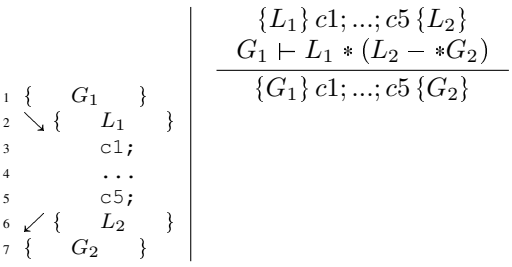
$$\frac{\dfrac{\dfrac{}{\forall x, (L'(x) - *G'(x)) \vdash L'(x_0) - *G'(x_0)}}{\forall x, (L'(x) - *G'(x)) * L'(x_0) \vdash G'(x_0)}}{\forall x, (L'(x) - *G'(x)) * \exists x, L'(x) \vdash \exists x, G'(x)}$$
$$\overline{\forall x, (L'(x) - *G'(x)) \vdash \exists x, L'(x) - *\exists x, G'(x)}$$

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.

6.4 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.



$$\frac{\{L_1\}\, c1; ...; c5\, \{L_2\} \qquad G_1 \vdash L_1 * (L_2 -\!\!* G_2)}{\{G_1\}\, c1; ...; c5\, \{G_2\}}$$

Figure 3. Localize and unlocalize in 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.

7. Ramification based on VST

7.1 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.

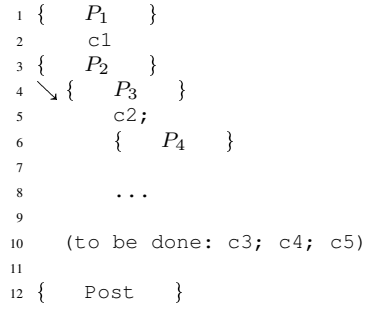
For example, when the proof goal is $\{P_1\}\, c1; c2\, \{P_5\}$, VST's user can apply some Hoare rule to get a triple for c1, e.g. $\{P_1\}\, c1\, \{P_2\}$. VST's tactic system then applies sequence rule automatically and the proof goal left to user will be $\{P_2\}\, c2\, \{P_5\}$. On user's view, 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).

7.2 Extend VST to Support Ramification

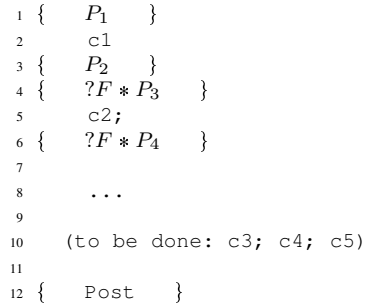
In order to extend VST to support ramification, we should enable people to write decorated programs with localize and unlocalize action. Our task here is to construct a proof in Coq's underlying logic from a decorated program, in which localize and unlocalize are involved. Moreover, our extension of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



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:



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

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.

8. 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.

9. 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.

10. 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 2015]. 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]

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.

11. 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.

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.

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

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. 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, 2015. ISSN 1661-8289. doi: 10.1007/s11786-014-0183-z. URL http://dx.doi.org/10.1007/s11786-014-0183-z.

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.

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