Certifying Graph-Manipulating C Programs via Localizations within Data Structures

Extended Abstract

Shengyi Wang
School of Computing, National University of Singapore

Qinxiang Cao
John Hopcroft Center, Shanghai Jiao Tong University

Anshuman Mohan
Yale-NUS College and School of Computing, National University of Singapore

Aquinas Hobor
Yale-NUS College and School of Computing, National University of Singapore

ACM Reference Format:
Shengyi Wang, Qinxiang Cao, Anshuman Mohan, and Aquinas Hobor. 2019. Certifying Graph-Manipulating C Programs via Localizations within Data Structures: Extended Abstract. In *Proceedings of Asian Symposium on Programming Languages and Systems (APLAS 2019)*. ACM, New York, NY, USA, **??** pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

We develop a comprehensive set of techniques to mechanically verify executable C programs that manipulate heap-represented graphs. The critical problem in this area is that separation logic does not work its usual miracles: graphs exhibit *deep intrinsic sharing*, causing the FRAME rule to fail.

We construct a modular and general setup for reasoning about abstract mathematical graphs (§**??**), and provide a key extension to separation logic and an improved way of representing abstract graphs concretely in the heap (§**??**). Our techniques are general, powerful, and scalable: we integrate them into the Verified Software Toolchain and produce Coq-checked proofs of correctness for seven classic graph-manipulating programs written in CompCert C. In the interest of space we discuss only our flagship example, a garbage collector for the CertiCoq project (§**??**). We provide brief comparisons with the state-of-the-art throughout.

1 A Framework for Abstract Graph Theory

When verifying the functional correctness of graph algorithms, it is natural to use graph theory to describe program behavior. We develop a new graph theory library that allows reasoning about a wide variety of mathematical graphs. Although real-world programs manipulate customized graphs with very specific properties, those properties can often be described as a combination of simpler properties. Our library takes advantage of this to allow the systematic construction of mathematical graphs from atomic parts that we provide out of the box. Figure **??** shows the architecture of our mathematical graph library.

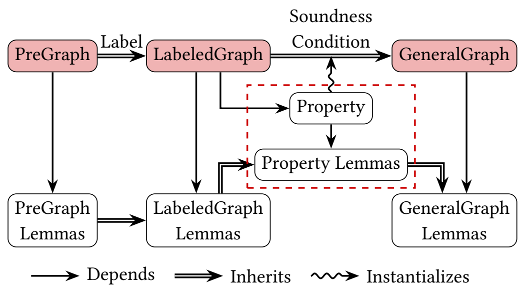

Figure 1. Structure of the Mathematical Graph Library

A PreGraph is a hextuple $(\mathcal{V}, \mathcal{E}, \mathtt{V}, \mathtt{E}, \mathtt{src}, \mathtt{dst})$. Arguments $\mathcal{V}$ and $\mathcal{E}$ are the underlying carrier types of vertices and edges. $\mathtt{V}$ and $\mathtt{E}$ are predicates over $\mathcal{V}$ and $\mathcal{E}$ that decide of *validity* in the graph. Finally, $\mathtt{src}$ and $\mathtt{dst} : \mathcal{E} \to \mathcal{V}$ map each edge to its source and destination. We pre-define many fundamental graph concepts on PreGraphs, including structures like *path*, predicates such as *is_cyclic* and *reachable*, operations such as *add_vertex* and *remove_edge*, and relations between PreGraphs such as *structurally_identical* and *subgraph*. These general concepts, together with around 500 derived lemmas, provide a solid foundation for more specific theorems needed in concrete verifications.

A LabeledGraph augments a PreGraph with custom *labels* of arbitrary type on vertices, edges, and on the graph as a whole. Many classic graph problems manipulate labels/weights, and so the need for LabeledGraph is unsurprising. A LabeledGraph smoothly inherits any properties and lemmas about its internal PreGraph.

PreGraphs and LabeledGraphs let us state and prove lemmas that follow from their constructions. However, modeling a real program's behavior calls for more specificity. For example, an algorithm may require that each vertex have exactly one out-edge and cycles be forbidden. Of course, these restrictions vary greatly by algorithm, so we do not want to bake them into our core definitions. We achieve this flexibility using GeneralGraphs, which augment LabeledGraphs by adding arbitrarily complex "soundness conditions", indicated in Figure **??** with a dashed border. Further, a GeneralGraph can seamlessly behave like its internal LabeledGraph or a PreGraph, thereby inheriting their lemmas. This combination of specificity and generality makes GeneralGraphs versatile. Moreover, we can create complicated soundness conditions by composing smaller reusable pieces, further enabling code sharing between algorithms.

Comparison with the state-of-the-art. Prior works in the verification of graph-manipulating programs have also built libraries such as ours, but they tend to suffer a few shortcomings: (1) they avoid linking with a real executable language, thus avoiding many of the subtleties that we have worked out; and (2) they are typically geared toward the verification of specific examples or classes of examples, meaning their reusability and modularity is limited.

2 Graphs on the Heap

The new LOCALIZE rule connects the "local" effect of a command c, *i.e.* transforming L_1 to L_2, with its "global" effect, *i.e.* from G_1 to G_2. LOCALIZE is a more general version of the well-known FRAME rule, which does the same task in the simpler case when $G_i = L_i * F$ for some frame F that is untouched by c. LOCALIZE and FRAME are coderivable, so we know our feet are firmly planted.

$$\textsc{Localize}\quad \frac{G_1 \vdash L_1 * R \qquad \{L_1\}\ c\ \{\exists x.\ L_2\} \qquad R \vdash \forall x.\ (L_2 -\!\!* G_2)}{\{G_1\}\ c\ \{\exists x.\ G_2\}}\ (\dagger)$$

$(\dagger)$ $\mathit{freevars}(R) \cap \mathsf{ModVar}(c) = \emptyset$

The standard approach to describing the layout of graphs on the heap is to rely on the Knaster-Tarski fixpoint and define graph as a recursive fixpoint. We show that this approach is flawed, and that a flat structure is better.

We start with the iterated separating conjunction or "big star", which is defined over lists and extended to sets. This allows us to define a better graph predicate, as follows:

$$\mathop{\circledast}_{\{l_1,\ldots,l_n\}} P \triangleq P(l_1) * P(l_2) * \ldots * P(l_n)$$

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

$$\mathsf{graph}(x,\gamma) \triangleq \mathop{\circledast}_{v \in \mathit{reachable}(\gamma,x)} v \mapsto \gamma(v)$$

Armed with these two tools, we go on to develop a second library, which allows intuitive reasoning about graphs as they are manipulated on the heap. Figure ?? shows the architecture of our spatial development. Once again, we pay attention to modularity and reuse. We support both the step-indexed and direct heap models. Out of the box, we provide an extensive library of lemmas about separation logic, $\circledast$, and graph. Further, we provide a number of lemmas that, in many common cases, handle the "ramification entailments" of LOCALIZE for us, thus making its use much simpler.

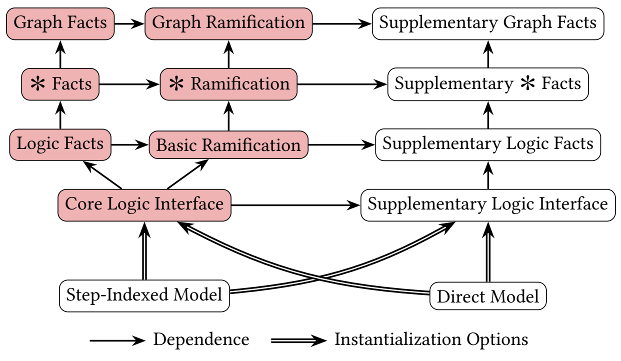

Figure 2. Infrastructure of ramification library

Comparison with the state-of-the-art. LOCALIZE is an improvement in two key ways: it allows support for existential quantifiers in postconditions and smoother treatment of modified program variables. Both of these prove helpful when reasoning about graphs on the heap. We show that recursively defined graph predicates, which are in common use, are a poor idea. We provide a flat definition, along with a proof—with the amount of precision required to convince Coq—that we still enjoy fold/unfold with our flat definition.

3 Certification of a Garbage Collector

The CertiCoq compiler's 400-line generationalgarbage collector, which is written in the spirit of the OCaml GC, is realistic and sophisticated, though not industrial-strength. Because CertiCoq uses OCaml's representation of blocks and values, the GC must support features such as 12 generations, variable-length memory objects, object fields that may be boxed or unboxed and must be disambiguated at runtime, and pointers to places outside the GC-managed heap.

It is an important example for our purposes because it shows that our techniques actually *scale* well beyond short toy programs. The manipulations that the GC performs are delicate and numerous enough that a brute force approach involving custom predicates is a tall task indeed. Happily, our two libraries and our LOCALIZE rule save the day.

Interestingly, we find and fix two bugs in the GC source code, and also discover two places where the semantics of Clight is unable to specify an OCaml-style GC such as ours. Although the Clight code is undefined, we prove that CompCert's compiler transformations preserve the necessary invariants for both operations.

Comparison with the state-of-the-art. Verifying GCs often involves customized predicates and onerous steps such as annotating x86 code by hand, writing the GC in Assembly, *etc.* Our framework allows us to work at a comfortably high level. Further, our GC ranks high in terms of complexity and real-world use: it has many of the complications of the industrial-strength OCaml GC, and it fits into CertiCoq, a real verified compiler.