#268 Certifying Graph-Manipulating C Programs via Localizations within Data Structures

Main Edit Your submissions (All) Search

✓ **Email notification**
Select to receive email on updates to reviews and comments.

PC conflicts
None

▶ **Other conflicts**

Submitted

Submission 6 Apr 2019 12:23:32am AoE · def32277

▼ Abstract

We develop powerful and general techniques to mechanically verify realistic programs that manipulate heap-represented graphs and related data structures with intrinsic sharing. We construct a modular and general setup for reasoning about abstract mathematical graphs and use separation logic to define how such abstract graphs are represented concretely in the heap. We upgrade Hobor and Villard's theory of ramification to support existential quantifiers in postconditions and to smoothly handle modified program variables. We demonstrate the generality and power of our techniques by integrating them into the Verified Software Toolchain and certifying the correctness of six graph-manipulating programs written in CompCert C, including a 400-line generational garbage collector for the CertiCoq project. While doing so, we identify two places where the semantics of C is too weak to define generational garbage collectors of the sort used in the OCaml runtime. Our proofs are entirely machine-checked in Coq.

▶ Authors (blind)

S. Wang, Q. Cao, A. Mohan, A. Hobor [details]

✓ **We intend to submit an artifact**

▶ Topics

OveMer RevExp

Review #268A	A	Y
Review #268B	B	Z
Review #268C	B	X

You are an **author** of this submission.

Edit submission **Add response**

Reviews in plain text

Review #268A

Overall merit	Reviewer expertise
A. Accept	**Y.** Knowledgeable

Paper summary

This paper concerns itself with verifying programs that manipulate graphs in memory. To achieve this goal, the authors introduce the concept of “localisation” to the Separation Logic-based reasoning that is typically used when verifying the effects of program statements on memory so that the unchanged parts (“outside the frame”) can be safely ignored. The localisation employed here allows one to reason about the local memory changes specific to a particular statement (which is typically easier) and then the paper presents a way to connect such local results to the global reasoning about the memory state of the program. The main trick is how to produce a statement about the affected vs unaffected memory so that it is useful enough to claim that local changes and separately the unaffected memory imply that the global changes are correctly made. The authors refer to this as “ramification entailment” but the explanations in the paper itself are surprisingly clear enough for even non-experts to grasp their meaning.

A running example of manipulating the disjoint set commonly used in Union Find algorithm is used to explain the approach. Further examples include the marking algorithm for spanning trees and most interestingly: a generational GC where a couple of new issues are discovered.

Assessment

I have thoroughly enjoyed reading this paper. Despite me being non-expert in the area, I found all the explanations to be very clear and self contained and the reader is guided through the detailed explanations of the contribution, motivations, and the issues involved. I would love to see this paper published in OOPSLA.

I would still work on the explanations regarding the core contribution: working out ramification entailments both in terms of their use in the running examples and your support for them described in 5.3.

Detailed comments

- Some of the language in the paper is very informal. While I personally liked it and it helps to guide the reader through, some people may not like it as much.
- This is optional but would help read all your code+verification examples: can you by chance make the "code lines" have light grey background to stand out more easily?

- In Section 2, can you please make the ramification entailment somehow more obvious in your running example in the code listing in Figure 1? I understand it is tight, but either more text on how one arrives at the two or some visual assistance in addition to red might help.
- Odd request but always makes me wonder, can you add to 7.2 some guestimate of "Human-Hours" it took you to verify the code?

Small Typos:

Line 44: "Second, we use"

Line 233: "as will be explained"

Line 336: "With careful engineering"

Line 478: "atop of it"

Line 590: at the end of P and separately Q expansion: do you mean Q instead of number 2?

Also, is Figure 7 actually referred to in the text somewhere? I can see it referred to in two footnotes but I don't think you refer to it in the text so it looks odd plonked there in the middle. :-)

Review #268B

Overall merit
B. Weak accept

Reviewer expertise
Z. Outsider

Paper summary

This paper presents a new separation logic based technique that can perform machine-checked verification of graph manipulating programs. One of the key challenges in using Hoare logic to prove such programs is how to deal with the implicit sharing that exists in graphs but not trees. The paper develops a notion of localization blocks that connects global predicates on a graph with local predicates on variables. The key idea is to choose a ramification frame that satisfies several important conditions. This technique, combined with mathematical and spatial graphs, enables the authors to be able to verify real C programs such as the CertiCoq garbage collector.

Assessment

Pros:

+ The paper targets an important problem -- how to verify graph-manipulating graphs in a modular way
+ The proposed techniques appear novel
+ Use of the technique helped the authors find real bugs!
+ Solid engineering effort

Cons:

- The presentation can benefit from a background section that familiarizes the reader with some of the key ideas in separation logic and other related works

Detailed comments

This is a relatively low confidence review: much of the paper concerns the ways in which modular verification can be done for graph manipulating programs, and it falls quite outside my expertise in the graph analytics and systems.

First off, the paper targets a very important problem: how to verify graph manipulating programs using separation logic. Graph is becoming increasingly popular in real-world analytical tasks, and hence being able to verify programs that manipulate graphs is highly commendable! The paper improves the state of the art by developing both new theories (e.g., proof techniques) and solid implementations that can work for real programs (such as a GC!)

One of the issues is that the presentation, although clear and not very difficult to follow in most cases, becomes quite involved even in the introduction. To make the paper accessible to a general audience, the authors should consider adding a background section that provides necessary information about separation logic and how existing works use it to verify tree manipulating programs. This section can also highlight the differences and challenges in verifying graph manipulating programs. Whoever is already familiar with the related work can skip this section while others would definitely be more appreciative if such a section can be added.

Review #268C

Overall merit	Reviewer expertise
B. Weak accept	X. Expert

Paper summary

This paper describes a general technique for verifying programs that manipulate graphs represented in the heap. The approach is based on Separation Logic, implemented in VCC, and put to practice on the verification of a GC algorithm that essentially consists of a simplified (yet sufficiently realistic) version of OCaml's GC algorithm. One key technical ingredient is the "localize" rule, a specialized version of the "ramified frame rule", which is itself a practical, convenient reformulation of the "frame rule".

Assessment

Strong points:

- The paper demonstrates the ability to verify a GC algorithms by reasoning at the level of abstraction of graphs, that's great.

- The paper's technology is smoothly integrated in a large framework for the effective verification of C code, namely VCC. Unlike many other approaches to program verification that consider a clean language and treat local variables as resources, VCC makes the choice of addressing all the complexity of C upfront, giving it a unique ability to verify general C code.

- The proposed representation of graphs is also slightly more general than what I have seen in prior work. The additional generality is useful to reason about graphs that are implemented in the heap but are not presented as a standalone, independent data structure as done when considering a graph algorithm all by itself. This possibility is very important for verifying complex software that embeds graph algorithms as a component of a larger algorithm.

Mitigated points:

- The "localize" rule appears to me as just a specialization of the ramified frame rule. At least, if following the variable as resources approach, the rule appears as just a particular instance. Now, in the VCC framework, dealing with variables and existential requires great care and effort. So maybe in that context the statement of the localize rule is a valuable contribution.

Weak points:

- The paper never really motivates the interest of considering "minimally small footprint", that is, why would one be interested to restrict a precondition to only the set of reachable vertices, when in the worst case all vertices might be reachable? Although there might be some philosophical interest to considering minimal footprints, from a practical perspective it is not obvious what the gain is. I suspect that, in most algorithms, the invariants must be strong enough to handle the worst case, so the same invariants would also hold when not all vertices are reachable. (I'd be happy to be proved wrong on this point, through a convincing example.)

- It is not entirely clear how far one can go with reasoning using overlapping conjunction. (Recall that the entire point of Separation Logic is to avoid overlaps.) The authors seem to take it for granted that it is necessary (at least, in their approach), although other work have verified similar algorithms without overlapping conjunction. Further explanations would be helpful.

- Related work section is lacking several pointers, and fails to given a proper comparison to related work on the formalization of graph algorithms. Details follows.

Due to these limitations, in its current state and without further information, I cannot give more than a B-score to this paper.

In the related work section, the part devoted to the "mechanized verification of graph theory" is missing the discussion of a number of closely related work. The formalization of graph algorithms is a central topic of the paper. Yet, only 10 lines are devoted to that aspect of the related work. These 10 lines only contains short citations, and do not provide any form of "comparison" per se. The discussion there is insufficient, in my opinion.

For one, recent work has tackled the formalization of graph algorithms that, from the perspective of graph theory, are far more involved than the algorithms treated in the present paper. (A GC is certainly a complicated beast, true, yet from the graph algorithm perspective, it essentially consists of a graph traversal with marking.)

- Peter Lammich's line of work includes formalizations of Ford-Fulkerson, Edmonds-Karp, Floyd-Warshall, Kruskal, .. https://link.springer.com/article/10.1007/s10817-017-9442-4 https://dblp.org/pers/hd/l/Lammich:Peter

- CFML's line of work, including a verification of Dijkstra's algorithm in Separation Logic (as old as 2011, published at ICFP), and more recently applied e.g., to a state-of-the-art incremental cycle cycle detection algorithm. http://gallium.inria.fr/~fpottier/publis/gueneau-jourdan-chargueraud-pottier-2019.pdf

- A collaborative work compares Tarjan's strongly connected components in Why3, Coq and Isabelle: https://hal.inria.fr/hal-01906155/document

There might be other (old or recent) papers that I am not aware of, further digging by the authors might be useful.

Admittedly, in the related work above, graph algorithms are not implemented in C but in languages that makes verification easier. But still, those papers could be mentioned, and some discussion of how the implementation language and how the graph representation in memory differ would be useful.

For two, there is even more to discuss specifically with respect to related work that also uses Separation Logic to formalize graph algorithms.

The present paper discusses for 2 pages the Union-Find example, but does not include a single line to compare against the prior formalization of Union Find in Separation Logic. The relevant cite is JAR'17 paper, available from: http://gallium.inria.fr/~fpottier/dev/uf/ (together with the Coq scripts). There, the graph is described as the iterated separating conjunction of the representation of each vertex, and there is a rule to isolate one given vertex from the others, so as to access its data. The verification of find there does not seem to be more complicated than that from Fig 1. I would like to understand precisely what are the benefits of using the localize rule as opposed to following the same approach as in that prior work.

Detailed comments

Rebuttal

Please provide in the response an udpated version of the paragraph that discusses related work on verification of graph algorithms, so that your text gets a chance to be reviewed.

Please also provide a motivation for how it does help to consider only the footprint of reachable nodes, as compared to the set of all vertices in the graph at hand.

Comments

The way labeled graphs are represented is by means of a function from edges to values. I suspect that this presentation is able, unlike many others, to handle multiple edges with different weights between the same pair of vertices. It might be worth pointing out if it is the case, as it is a nice feature.

The definitions on graphs involve standard notions of sets, in particular finiteness, and fold over a set. However, the authors stick to using Coq predicates everywhere, which is a possible (and natural) realization of sets in Coq, but lacks somewhat the abstraction and the convenient notation associated with conventional mathematical practice.

Presentation

- abstract: mentions "six" algorithms being verified, but the paper contains at no point a clear list with six items (line 76 contains three, lines 84 refers to the GC algorithm).
- line 485: at this point, it is not obvious at first sight what is the type of src and dst. Could be worth writing that it has type "edge_type -> vertex_type". But then, is the value of the functions irrelevant outside of the domain E? Is it arbitrary or do you impose a specific value?
- line 530: I wasn't able to guess the role of variable M_v. I think the presentation would be cleaner if defining once and forall the notion of "finite set" and of "fold over a finite set". E.g. finite (E:A->Prop) = exists (L:list A), x \in L <-> x \in E.
- line 654: "reach" is undefined. Is it "reachable"? It would be nice to have the definition in fig6, as it is essential to the statement of the fold/unfold lemmas.
- line 904, is it proper to use "Floyd" which is a last name relevant in the context of graph algorithms and verification, as a subject? Wouldn't it be more appropriate to write "The Floyd module"?
- page 29 in the appendix the figure overflows.

Response

Edit

Author [Anshuman Mohan <anshumanmohan@live.com>] Tuesday 11 Jun 2019 9:10:17pm AoE

2150 words

Response submitted.

High Level Response

Thank you for your time and careful attention to our paper. We appreciate all of the careful suggestions to help us improve the presentation. We also appreciate your heartening enthusiasm for our work. We respond to your specific comments and instructions as follows.

Response to Review #268A

> I would still work on the explanations regarding the core contribution: working out ramification entailments both in terms of their use in the running examples and your support for them described in 5.3.

We agree that this would be useful to add and will do so.

> Can you by chance make the "code lines" have light grey background to stand out more easily? Can you please make the ramification entailment somehow more obvious in your running example in the code listing in Figure 1?

A light grey background or similar is an excellent idea. We will also try to make the ramification entailments clearer.

> Odd request but always makes me wonder, can you add to 7.2 some guestimate of "Human-Hours" it took you to verify the code?

It’s difficult to say for the project as a whole. The certified garbage collector took eight months.

Response to Review #268B

> The authors should consider adding a background section that provides necessary information about separation logic and how existing works use it to verify tree manipulating programs. Whoever is already familiar with the related work can skip this section while others would definitely be more appreciative if such a section can be added.

We will provide some additional background material on “vanilla” separation logic in the early part of the paper. Labeling as you suggest is a good idea.

Response to Review #268C

> I suspect that this presentation is able, unlike many others, to handle multiple edges with different weights between the same pair of vertices.

Yes, we support multiple edges between a pair of vertices, with each edge having its own label.

> Please also provide a motivation for how it does help to consider only the footprint of reachable nodes, as compared to the set of all vertices in the graph at hand.
>
> It is not entirely clear how far one can go with reasoning using overlapping conjunction.

Our development supports both “whole-graph” styles (used, for example, in our union-find example) and “local graph” styles (with overlapping conjunction, used, for example, in our mark-graph and mark-dag examples). Having both approaches available (to the best of our knowledge, no other tool can support both) gives verifiers options to choose the technique they think is best-suited for the specific algorithm under consideration. Having more tools available strikes us as intrinsically useful.

One advantage of a “local graph” style is that the proofs tend to behave more like traditional SL proofs, e.g. by “unfolding” an inductive predicate and recursing, so some human verifiers prefer this style. In a somewhat related vein, we have done some preliminary experiments that indicate that automated tools such as HIP/SLEEK may prefer a “local graph” style for exactly this reason: they fit within the kind of reasoning paradigm already understood by the tool.

[Hobor and Villard 2003] gives a number of examples using the overlapping conjunction, including a (highly simplified) Cheney-style GC (in which it is used for example to “drop the garbage on the floor”). We are not aware of any subtle issues or weaknesses with their approach other than those discussed and solved in §5. In a C-like memory model, the “skewing” issue requires the addition of alignment instructions as in lines 1-2 of figure 3.

As mentioned in the related work, [Gardner et al. 2012] and [Raad et al. 2015] have both used the overlapping conjunction / a more local specification style as well.

> The "localize" rule appears to me as just a specialization of the ramified frame rule. At least, if following the variable as resources approach, the rule appears as just a particular instance. Now, in the VCC framework, dealing with variables and existential requires great care and effort. So maybe in that context the statement of the localize rule is a valuable contribution.

The treatment of local variables we use is unnecessary in settings that allow variables-as-resource. However, the proper handling of existentials in postconditions is still helpful for developing clean proofs. For example, consider a call to a sorting algorithm inside a localization block. Suppose the sort function is specified by:

```
Precondition: list(x,lst)
Post-Condition: exists lst', list(x,lst') /\ ordered lst' /\ permutation lst lst'
```

Our rule will then allow us to unlocalize and keep the existential.

> I would like to understand precisely what are the benefits of using the localize rule as opposed to following the same approach as in [Charguéraud and Pottier 2015, 2019].

Localize is a general rule to "zoom in" and reason about pointer-manipulations of various flavors. For example, the examples Find (figure 1) and Forward (figure 9) use different graphs (uf_graph and gc_graph respectively, constructed as described in §4.1) and very different representations of spatial memory, but they employ the exact same Localize rule (i.e. ↘ ↙ blocks), and indeed can reuse lemmas from the ramification library, all of which are independent from any specific algorithm. We favor this generality over custom rules to isolate graph parts (vertices or subgraphs) for individual algorithms.

Please see our current draft of our related work section below for further comparison.

> Related work section is lacking several pointers, and fails to give a proper comparison to related work on the formalization of graph algorithms.

As requested, here is our newest draft for our related work section (sections marked "as submitted" are unchanged from our submission version).

RELATED WORK

Comparison with [Hobor and Villard 2013].

(as submitted)

Other pen-and-paper verification of graph algorithms and/or $\uplus *$.

[Yang 2001]'s verification of the Schorr-Waite algorithm is a landmark in the early separation logic literature. [Bornat et al. 2004] gave an early attempt to reason about graph algorithms in separation logic in a more general way. [Krishnaswami 2011] provided the first separation logic proof of union-find.

[Reynolds 2003] was the first to document the overlapping conjunction $\cup*$, albeit without any strategy to reason about it using Hoare rules. [Gardner et al. 2012] were the first to reason about a program using $\cup*$ in Javascript. [Raad et al. 2015] used $\cup*$ within their CoLoSL program logic to reason about a concurrent spanning algorithm using a kind of "concurrent localization".

Machine-checked verification of graph algorithms.

A decade after Yang verified Schorr-Waite on paper, [Leino 2010] automated its verification in Dafny. [Sergey et al. 2015] verified a concurrent spanning tree algorithm, and moreover developed mechanized Coq proofs. Their algorithm was written in FCSL, a monadic DSL that combines effectful operations with pure Coq expressions; FSCL cannot be executed. [Chen et al. 2018] compared how three provers (Coq, Isabelle, and Why3) can verify Tarjan's strongly-connected component algorithm written in the native language of each of the tools. Because these are written in the native languages of a proof assistant, they avoid "real-world" language concerns such as memory models and overflow.

[Lammich and Neumann 2015] extended the Isabelle Refinement Framework to verify a range of DFS algorithms via stepwise refinement. [Lammich and Sefidgar 2019] extended this further and presented verifications of the correctness and time complexity of the Edmonds-Karp and push-relabel algorithms. Lammich et al.'s work yields very readable proofs of classic textbook algorithms by using the Isar language.

[Charguéraud 2011] used his CFML tool to Coq-verify an OCaml implementation of Dijkstra. [Guéneau et al. 2019] extended CFML and verified the correctness and time complexity of a modified version of the BFGT cycle-detection algorithm. The graph algorithms verified in CFML tend to be "graph theory" in flavour, whereas the algorithms we have verified in this paper tend to have more of a "systems" flavor; this difference is partially explained by the fact that code written in ML can take advantage of its high-level design, whereas the "interesting" use for C code is to handle grungy systems tasks. For example, references in ML cannot be null and do not support pointer arithmetic; of course both are possible – and lead to nontrivial complications – in C. Accordingly, the CFML proofs benefit from ML's cleaner computational model; our verifications are in C so we must contend with C's memory model, pointer arithmetic, significant scope for undefined behavior, and so forth.

[Charguéraud and Pottier 2015, 2019] used CFML to verify the correctness and time complexity of union-find. Their work is an interesting counterpoint to ours because, while it maintains an abstraction between the client and the internal mathematical/spatial facts that the client need not know, it does not appear to maintain a separation between the mathematical and spatial facts themselves, as we do in §4 and §5. Our modular method let us verify an alternate version of union-find that uses an array of vertices rather than individually heap-allocated nodes. This secondary verification then used *exactly* the same mathematical proof of functional correctness despite the radically different layout of spatial memory. Our work does not verify the time complexity of union-find; when we attempted to prove the necessary amortisation bounds we ran into an overflow issue: it was impossible to prove that the rank would not exceed max_int because the CompCert memory model does not place a bound on the total number of allocations. Informally, this overflow is impossible in practice because no computer has 2^(2^64) bytes of memory, which would be required for this overflow to occur, but Coq remains unconvinced. Charguéraud and Pottier acknowledged and sidestepped this issue by representing rank using the Coq type Z, which is not an option we could use given the end-to-end nature of the VST + CompCert toolchain.

Verification tools in Coq.

Our work interacts with the Floyd verification module within the Verified Software Toolchain (VST) [Appel et al. 2014]. The Floyd module uses tactics to enable the separation-logic verification of CompCert C programs. VST connects to the CompCert certified C compiler [Leroy 2006], and thus has no gaps or admits between the verified source code and the eventual assembly code [Appel 2011].

Charge! likewise uses Coq tactics to work with a shallow embedding of higher order separation logic, but focuses on OO programs written in Java/C# [Bengtson et al. 2012]. Iris Proof Mode provides a similar framework for higher-order concurrent reasoning in Coq [Krebbers et al. 2017].

CFML enables the verification of OCaml programs by reasoning about their “characteristic formulae” in separation logic using Coq [Charguéraud 2010, 2011]. CFML has been used to verify a range of functional and imperative programs, including some graph-related algorithms as discussed above. [Charguéraud and Pottier 2015, 2019] extended CFML to reason about time credits. [Guéneau 2017] indicates that CFML is exploring a connection with the certified CakeML compiler.

The tools above require substantial human guidance. A more automated approach to verification of low level programs using Coq is the Bedrock framework [Chlipala 2011].

Other verification tools.

Many more-automated verification tools also use separation logic in a forward reasoning style. Smallfoot [Berdine et al. 2005], jStar [Distefano and Parkinson 2008], HIP/SLEEK [Chin et al. 2010], and Verifast [Jacobs et al. 2011] are landmarks at various points on the expressibility-automatability spectrum. KeY [Beckert et al. 2007] and Dafny [Leino 2010] are verifiers that are not based on separation logic. KeY uses an interactive verifier while Dafny pursues automation with Z3 [de Moura and Bjørner 2008].

Mechanized mathematical graph theory.

There is a long history, going back at least 28 years, of mechanized reasoning about mathematical graphs [Wong 1991]. The most famous mechanically verified “graph theorem” is the Four Color Theorem [Gonthier 2005]; however the development actually uses hypermaps instead of graphs. In general most “mathematical graph” frameworks in the literature were not used to verify real code [Wong 1991; Chou 1994; Yamamoto et al. 1995; Butler and Sjogren 1998; Yamamoto et al. 1998; Tamai 2000; Duprat 2001; Ridge 2005; Nipkow et al. 2006; Nordhoff and Lammich 2012], for which they seem unsuitable. Verifying real code requires delicate concepts such as “removing a subgraph”, “null nodes”, and “parallel edges”, and one of our contributions is that our framework is general enough to support such verification. [Noschinski 2015b] built a graph library in Isabelle/HOL whose formalization is the closest to ours, e.g. supporting graphs with labeled and parallel arcs. Beyond being in Coq, our setup supports at least three features beyond Noschinski’s: reasoning about incomplete graphs (as discussed in §4.1 using figure 5), labeling the graph as a whole (used, for example, in the garbage collector to store metainformation about the number and location of the generations), and our modular typeclass-supported “graphs with properties” setup in General Graph (as described in §4.2). [Dubois et al. 2015; Noschinski 2015a] used proof assistants to design verifiable checkers for solutions to graph problems. [Bauer and Nipkow 2002; Yamamoto et al. 1995] used an inductive encoding of graphs to formalize planar graph theory.

Verification of garbage collection algorithms.

(as submitted)