OOPSLA 2019 Paper #268 Reviews and Comments

===========================================================================

Paper #268 Certifying Graph-Manipulating C Programs via Localizations

within Data Structures

Review #268A

===========================================================================

Overall merit

-------------

A. Accept

Reviewer expertise

------------------

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

-------------

B. Weak accept

Reviewer expertise

------------------

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.