Foundationally Sound Annotation Verifier via Control Flow Splitting

Litao Zhou
Shanghai Jiao Tong University
Shanghai, China
ltzhou@sjtu.edu.cn

Abstract

We propose VST-A, a foundationally sound program verifier for assertion annotated C programs. Our approach combines the benefits of interactive provers as well as the readability of annotated programs. VST-A analyzes control flow graphs and reduces the program verification problem to a set of *straightline Hoare triples*, which correspond to the control flow paths between assertions. Because of the rich assertion language, not all reduced proof goals can be automatically checked, but the system allows users to prove residual proof goals using the full power of the Coq proof assistant.

CCS Concepts: • **Theory of computation → Program verification**; *Hoare logic*.

Keywords: Program Verification, Annotated Programs, Coq

ACM Reference Format:
Litao Zhou. 2022. Foundationally Sound Annotation Verifier via Control Flow Splitting. In *Companion Proceedings of the 2022 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH Companion '22), December 5–10, 2022, Auckland, New Zealand*. ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3563768.3563956

1 Background

Many program verification tools have been developed and used in different ways, with their own advantages.

Interactive verification tools (such as VST [1], Iris [7, 8]) are based on interactive theorem provers (such as Coq [4]). One benefit of those tools is that they are *foundationally sound* (i.e., have a formal proof w.r.t. the language's operational semantics in the proof assistant). The rich language of theorem provers also makes interactive verification tools powerful in verifying real-world programs.

Annotation verifiers allow programmers to add assertion annotations directly into the code. Many verifiers work by reducing annotated programs to SMT entailments [2, 5, 6, 9, 11]. Compared with proof scripts in a theorem prover, *writing annotations is a much more straightforward way of demonstrating correctness of a program.* Even proofs written completely in an interactive prover are often presented in research papers as annotated programs. For example, Figure 1 shows a C program and its functional correctness proof [1]. The annotation on line 3 describes the specification. The assertion on line 5 states an invariant in the loop, which is the main idea of the correctness proof. Despite being succinct in describing correctness, annotation verifiers suffer from restricted assertion and proof languages. Though stapling those tools with external proof systems or solvers is possible [3], a common foundational soundness proof is missing.

2 Motivation

The goal of this research is to allow users to verify a program by writing readable assertion annotations, while retaining the benefits of interactive tools, such as rich assertion languages, flexible proof strategies, and most importantly, foundational soundness.

Existing works that build annotation verification into interactive provers [10, 13] use *tactic-based proof strategy* designs. A Hoare triple will be reduced to smaller proof goals by automatically applying a series of proof tactics. However, such decomposition is often not flexible enough. For example, a fixed-location loop invariant is always required to apply the Hoare logic rule for loops. An input program like Figure 1 is not accepted. Moreover, tactic based proof strategies are vulnerable to changes. When a user makes a change to a program, the entire proof needs to be recompiled.

Unlike previous tools, verification in VST-A is based on a *computational* proved-sound reduction function, so that assertions can be inserted anywhere in a program, and changes to the program only require recompiling the changed part.

3 Approach

We build VST-A, a foundationally sound verifier, based on VST in Coq. We illustrate VST-A's workflow in Figure 4:
(1) Users provide a C program with assertion annotations like Figure 1. Our front-end parser converts it into ClightA, an AST language for assertion annotated C programs.

SPLASH Companion '22, December 5–10, 2022, Auckland, New Zealand

ACM ISBN 978-1-4503-9901-2/22/12...$15.00
https://doi.org/10.1145/3563768.3563956

```
struct list {unsigned head; struct list *tail;};
struct list *reverse (struct list *p) {
  /*@ With l, Require ll ([[p]], l) Ensure ll ([[ret]], rev(l)) */
  struct list *w, *t, *v; w = NULL; v = p;
  while (v) { /*@ Assert ∃ l1 c x l2'. l = rev(l1) x l2' ∧ [[v]] ↦ (x, c) * ll ([[w]], l1) * ll (c, l2') */
    t = v->tail; v->tail = w; w = v; v = t; }
  return w; }
```

Figure 1. Annotated C program for verifying linked-list reversal

(1) $\forall\, l$,
$\{\ ll([\![p]\!], l)\ \}$
w = NULL; v = p; **assume** v;
$\{\ \exists\, l_1\ c\ x\ l_2'.\ l = rev(l_1)\ x\ l_2' \wedge [\![v]\!] \mapsto (x, c) * ll([\![w]\!], l_1) * ll(c, l_2')\ \}$
(2) $\forall\, l$,
$\{\ ll([\![p]\!], l)\ \}$
w = NULL; v = p; **assume** !v; ret = w;
$\{\ ll([\![ret]\!], rev(l))\ \}$
(3) $\forall\, l\ l_1\ c\ x\ l_2'$,
$\{\ l = rev(l_1)\ x\ l_2' \wedge [\![v]\!] \mapsto (x, c) * ll([\![w]\!], l_1) * ll(c, l_2')\ \}$
t = v->tail; v->tail = w; w = v; v = t; **assume** v;
$\{\ \exists\, l_1\ c\ x\ l_2'.\ l = rev(l_1)\ x\ l_2' \wedge [\![v]\!] \mapsto (x, c) * ll([\![w]\!], l_1) * ll(c, l_2')\ \}$
(4) $\forall\, l\ l_1\ c\ x\ l_2'$,
$\{\ l = rev(l_1)\ x\ l_2' \wedge [\![v]\!] \mapsto (x, c) * ll([\![w]\!], l_1) * ll(c, l_2')\ \}$
t = v->tail; v->tail = w; w = v; v = t; **assume** !v; ret = w;
$\{\ ll([\![ret]\!], rev(l))\ \}$

Figure 2. Straightline Hoare triples for verifying linked-list reversal

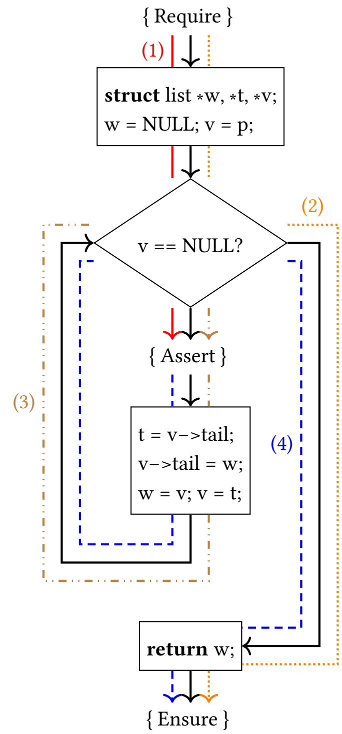

Figure 3. Control flow graph for reverse

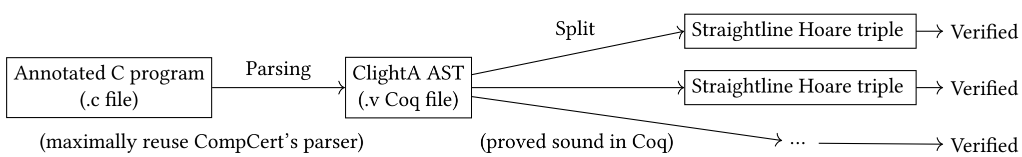

Figure 4. Verification workflow in VST-A

(2) Next, a *split function* accepts a ClightA program and its pre-/post-conditions as input, and returns a set of straightline Hoare triples, which contain only basic statements and no control flow structures. Figure 2 shows the split result of Figure 1. The functionality of this function is natural; it computes all the control flow paths that are separated by assertion annotations in the source program, as illustrated by Figure 3. We carefully design this function to deal with logical variables in the assertions, so that rich functional correctness properties expressed by existential quantifiers can be safely transformed.
(3) Finally, users are left to prove each straightline Hoare triple in the split result, which can be largely automated by existing VST forward symbolic execution tactics. With all of the paths verified, the VST-A soundness theorem ensures the correctness of the original program.

[1]This is a separation logic [12] proof. $ll([\![p]\!], l)$ is a separation logic predicate that asserts on the location referenced by p stores a linked list of l.

The primary challenge of building VST-A is establishing its foundational soundness. We implement the split algorithm as a Coq function, and formally prove its soundness as a Coq theorem. We find that the *conjunction rule*:

$$\{P\}\ c\ \{Q_1\} \wedge \{P\}\ c\ \{Q_2\} \Rightarrow \{P\}\ c\ \{Q_1 \wedge Q_2\}$$

appears necessary in this soundness proof. In the current design of VST-A, we focus on sequential program verification and derive this conjunction rule from the VST program logic without ghost updates, and thus obtain the end-to-end soundness of VST-A.

With the conjunction rule, the soundness proof of VST-A split algorithm is purely logical. We believe that a similar development can be applied to design other Hoare-style annotation verifiers for imperative languages.

Acknowledgments

This research is supervised by Qinxiang Cao and supported by NSFC under Grant No. 61902240.

References

[1] Andrew W. Appel. 2011. Verified software toolchain (Invited talk). In *Lecture Notes in Computer Science*, Vol. 6602 LNCS. Springer, Berlin, Heidelberg, 1–17. https://doi.org/10.1007/978-3-642-19718-5_1

[2] Patrick Baudin, François Bobot, David Bühler, Loïc Correnson, Florent Kirchner, Nikolai Kosmatov, André Maroneze, Valentin Perrelle, Virgile Prevosto, Julien Signoles, and Nicky Williams. 2021. The dogged pursuit of bug-free C programs: The Frama-C Software Analysis Platform. , 56–67 pages. https://doi.org/10.1145/3470569

[3] Sylvie Boldo, François Clément, Jean-Christophe Filliâtre, Micaela Mayero, Guillaume Melquiond, and Pierre Weis. 2014. Trusting computations: A mechanized proof from partial differential equations to actual program. *Computers and Mathematics with Applications* 68, 3 (2014), 325–352. https://doi.org/10.1016/j.camwa.2014.06.004

[4] Pierre Boutillier, Stephane Glondu, Benjamin Grégoire, Hugo Herbelin, Pierre Letouzey, Pierre-Marie Pédrot, Yann Régis-Gianas, Matthieu Sozeau, Arnaud Spiwack, and Enrico Tassi. 2014. Coq 8.4 Reference Manual. (jul 2014). https://hal.inria.fr/hal-01114602

[5] Wei Ngan Chin, Cristina David, Huu Hai Nguyen, and Shengchao Qin. 2012. Automated verification of shape, size and bag properties via user-defined predicates in separation logic. *Science of Computer Programming* 77, 9 (aug 2012), 1006–1036. https://doi.org/10.1016/j.scico.2010.07.004

[6] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. 2011. VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In *Lecture Notes in Computer Science*, Vol. 6617 LNCS. 41–55. https://doi.org/10.1007/978-3-642-20398-5_4

[7] R. A.L.F. Jung, Robbert Krebbers, Jacques Henri Jourdan, Aleš Bizjak, Lars Birkedal, and D. E.R.E.K. Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming* 28 (2018), 73. https://doi.org/10.1017/S0956796818000151

[8] Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017. Interactive proofs in higher-order concurrent separation logic. *ACM SIGPLAN Notices* 52, 1 (jan 2017), 205–217. https://doi.org/10.1145/3009837.3009855

[9] K. Rustan M. Leino. 2010. Dafny: An automatic program verifier for functional correctness. In *Lecture Notes in Computer Science*, Vol. 6355 LNAI. 348–370. https://doi.org/10.1007/978-3-642-17511-4_20

[10] Ike Mulder, Robbert Krebbers, and Herman Geuvers. 2022. Diaframe: Automated Verification of Fine-Grained Concurrent Programs in Iris. (2022), 16. https://doi.org/10.1145/3519939.3523432

[11] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2017. Viper: A verification infrastructure for permission-based reasoning. In *Dependable Software Systems Engineering*. Vol. 9583. Springer Verlag, 104–125. https://doi.org/10.3233/978-1-61499-810-5-104

[12] John C. Reynolds. 2002. Separation logic: A logic for shared mutable data structures. In *Proceedings - Symposium on Logic in Computer Science*. 55–74. https://doi.org/10.1109/lics.2002.1029817

[13] Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. 2021. RefinedC: Automating the foundational verification of C code with refined ownership types. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 158–174. https://doi.org/10.1145/3453483.3454036