A Developer-oriented Hoare Logic

Holger Gast

Wilhelm-Schickard-Insitut für Informatik
University of Tübingen
gast@informatik.uni-tuebingen.de

1 Motivation

Even with current automated reasoning technology, full functional verification requires human interaction to guide the proof: assignments to ghost variables (e.g. [1]) or intermediate assertions (e.g. [17]) need to be provided, and sometimes the prover's deductions need to be examined in detail (e.g. [1, §7],[13]). Indeed, some authors have argued that the developer's understanding will be necessary regardless of advances in automation (e.g. [17, §7.2][8, §4.3][2, §1.2]).

For effective interaction, the user has to understand the generated verification conditions. While it is possible to relate them back to the source code by suitable highlights and annotations (e.g. [4, 9]), this approach does not cover the verification conditions themselves. For instance, the conditions usually express side-effects in the program by substitutions (e.g. [4, §4.2]), which in a weakest precondition calculus relate only loosely to a developer's view on the code.

We therefore propose to design the Hoare logic and verification environment itself to increase the developer's understanding of the verification process. Taking lightweight separation [5, 6, 7] as the basis, this paper presents a suite of verification tools developed around that method and their application to case studies. Although the language treated is a C dialect with a finite byte-addressed memory model, even involved algorithms like Schorr-Waite graph traversal lead to natural and readable proof obligations. Since assertions employ classical first-order (or higher-order) logic, existing automated reasoners apply. The tools are developed as a conservative extension of Isabelle/HOL, which ensures their correctness.

2 Developer-Orientated Verification Conditions

Our approach obeys the following principles to achieve developer-oriented verification. While some have been proposed in the literature individually, their combination leads to a verification process that closely resembles stepping through a program using a symbolic debugger, thus appealing to a developer's natural understanding of the code.

Forward Reasoning Developers usually discuss their code in terms of its eventual execution: they refer to parameters and the initial state, state transformations and computations, and then deduce that the final result will be the one expected. Our Hoare logic and tool suite consequently transform a statement's pre-condition into a readable post-condition, with the special characteristic that the result will not refer to the pre-state (except through logical variables), but only to the "current" post-state of the execution. The user can then argue that the generated post-condition implies the one given in the specification. While several studies have recognized the benefits of forward reasoning (e.g. [10]), their employed logics lack the other properties discussed below.

Classical (First-order) Logic Acceptance of verification can be increased by using a logic that developers know from their education, i.e. classical first-order logic (or higher-order logic, if desired). As a by-product, effective reasoners become available. While recent studies have shown that separation logic can be used for fully automatic proofs for particular classes of algorithms [3], even slightly involved examples need interactive guidance [16], and for complex algorithms, extensive manual proofs are required [14, 15, 11].

Support for Intuitive Disjointness Reasoning Developers usually find arguments about the disjointness of memory regions "obvious" – they draw pointer diagrams and argue about geometrical disjointness. Since lightweight separation supports such reasoning [6], the verification tools preserve those parts of the pre-condition that "obviously" remain unchanged in the computed post-condition, thus reflecting the developer's understanding in this important aspect. Contrary to other approaches (e.g. [4, §4.2]), the side-effects are thus removed, which results in natural and readable post-conditions.

Natural Execution Paths Standard Hoare logics split verification conditions between different execution paths, such that case distinctions, but also boolean short-circuit operators, induce separate verification conditions. The developer, on the other hand, writes these constructs in order to continue afterwards regardless of the encountered case. Instead of treating the execution paths separately, we therefore join their resulting post-conditions, unless the users chooses splitting explicitly.

Guided Verification Verification condition generation usually reduces correctness to a (usually large) number of verification conditions in a black-box step. When automatic provers fail, the user has to inspect them one-by-one nevertheless. We let the user guide the verification process at the source-code level instead. Other studies that have suggested this approach (e.g. [12]) have relied on backward-style Hoare rules, thus destroying the potential benefits.

3 Contributions

While the technical basis of lightweight separation – disjointness reasoning and automatic unfolding of memory layouts – has been discussed before [5, 6], their direct application suggested in [5] does not yield a practical verification environment. This paper discusses for the first time the tools developed around the given basis.

Normal forms for assertions We reduce the reasoning to a normal form of assertions, where all existentials – or ghost variables – are bound at the outermost level, intermediate results are substituted eagerly, and assertions only refer to the current memory state. Forward reasoning can then be expressed as an application a Hoare rule for the given programming language construct, followed by the normalization of the obtained post-condition. A detailed examination of normalization clarifies and facilitates the reasoning process discussed only loosely in [5] and enables a more efficient implementation.

One-pass simplification of side-effects Using Isabelle's simplifier directly for disjointness reasoning [5] does not scale even to medium-sized examples. A new rewriting procedure has been implemented which requires only one (top-down) pass through the assertion to remove all side-effects and caches memory regions already located for later re-use.

Automatic proofs about accessed regions When users define a new predicate about memory, they have to prove a theorem about the memory region it accesses [5]. The tool suite provides tactics to perform these proofs automatically, including the case of recursive structures such as linked lists or binary trees. Since the overhead for introducing new predicates is thus minimal, users are encouraged to employ application-specific abstractions in specifications and proofs.

Guided Verification The `step` tool orchestrates the other tools during forward reasoning and thus provides a convenient access point for the user. It simulates a debugger by continuing until the next statement or some boolean condition is found. In the latter case, the user can manipulate the pre-condition interactively (inside Isabelle) to exploit the information obtained from the boolean test before proceeding. We will also discuss commonly occurring patterns of reasoning that can serve as proof strategies for lightweight separation and enable, as a future research direction, fully automatic verification.

4 Examples

We demonstrate that the goals of a developer-oriented Hoare logic have been fulfilled by examining the verification process, and most importantly the computed post-conditions, for a number of examples: algorithms on linked lists (reverse, append); arrays (binary search, comprehension/filter); binary search trees; Schorr-Waite graph traversal.

References

[1] A. Banerjee, M. Barnett, and D. A. Naumann. Boogie meets regions: A verification experience report. In N. Shankar and J. Woodcock, editors, *VSTTE'08*, volume 5295 of *LNCS*, pages 177–191. Springer, 2008.

[2] S. Böhme, M. Moskal, W. Schulte, and B. Wolff. HOL-Boogie – an interactive prover-backend for the Verifying C Compiler. *J. of Automated Reasoning*, 2009.

[3] M. Botincan, M. Parkinson, and W. Schulte. Separation logic verification of C programs with an SMT solver. In *4th International Workshop on Systems Software Verification (SSV 2009)*, Electronic Notes in Theoretical Computer Science. Elsevier Science B.V., 2009.

[4] L. Burdy, A. Requet, and J.-L. Lanet. Java applet correctness: A developer-oriented approach. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME*, volume 2805 of *LNCS*, pages 422–439. Springer, 2003.

[5] H. Gast. Lightweight separation. In O. Ait Mohamed, C. Munoz, and S. Tahar, editors, *Theorem Proving in Higher Order Logics 21st International Conference, TPHOLs 2008*, volume 5170 of *LNCS*. Springer, 2008.

[6] H. Gast. Reasoning about memory layouts. In A. Cavalcanti and D. Dams, editors, *FM 2009: Formal Methods,Second World Congress*, volume 5850 of *LNCS*. Springer, 2009.

[7] H. Gast and J. Trieflinger. High-level reasoning about low-level programs. In M. Roggenbach, editor, *Automated Verification of Critical Systems 2009*, volume 23 of *Electronic Communications of the EASST*. EASST, 2009.

[8] C. Hawblitzel and E. Petrank. Automated verification of practical garbage collectors. *SIGPLAN Not.*, 44(1):441–453, 2009.

[9] K. R. M. Leino, T. Millstein, and J. B. Saxe. Generating error traces from verification-condition counterexamples. *Sci. Comput. Program.*, 55(1-3):209–226, 2005.

[10] A. McCreight. Practical tactics for separation logic. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *Theorem Proving in Higher Order Logics 22nd International Conference (TPHOLs 2009)*, 2009.

[11] A. McCreight, Z. Shao, C. Lin, and L. Li. A general framework for certifying garbage collectors and their mutators. *SIGPLAN Not.*, 42(6):468–479, 2007.

[12] J. Meyer and A. Poetzsch-Heffter. An architecture for interactive program provers. In S. Graf and M. Schwartzbach, editors, *TACAS '00, Tools and Algorithms for the Construction and Analysis of Software*, volume 276 of *LNCS*, pages 63–77, 2000.

[13] M. Moskal. Programming with triggers. In *SMT '09: Proceedings of the 7th International Workshop on Satisfiability Modulo Theories*, pages 20–29, New York, NY, USA, 2009. ACM.

[14] M. O. Myreen. *Formal verification of machine-code programs.* PhD thesis, University of Cambridge, Dec. 2009. UCAM-CL-TR-765.

[15] H. Tuch. Structured types and separation logic. In *3rd International Workshop on Systems Software Verification (SSV 08)*, Feb. 2008.

[16] T. Tuerk. A formalisation of Smallfoot in HOL. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *Theorem Proving in Higher Order Logics 22nd International Conference (TPHOLs 2009)*, volume 5674 of *LNCS*. Springer, 2009.

[17] K. Zee, V. Kuncak, and M. Rinard. Full functional verification of linked data structures. *SIGPLAN Not.*, 43(6):349–361, 2008.