

# Towards Verification by Symbolic Debugging

Holger Gast\*

\*Wilhelm-Schickard-Institut für Informatik  
University of Tübingen  
Sand 13, 72076 Tübingen  
gast@informatik.uni-tuebingen.de

## Abstract

Semi-automatic or interactive software verification requires the programmer to understand the generated proof obligations by connecting them to the source code being verified. A major difficulty lies in the backward-style reasoning employed by most programming logics. We propose that Hoare-style verification with forward reasoning is feasible and useful. The verification process appears to the user as “symbolic debugging”, in which the pre-condition of a Hoare triple captures the “current state” and evolves to reflect the side-effects encountered during execution. As a result, programmers can apply their operational understanding of the program to the solving the arising proof obligations.

## 1 Introduction

Many verification tools highlight the connection between the verification conditions and the source code to facilitate verification: Boogie (Barnett et al. (2006)) reports failed proofs as annotations; JACK (Burdy et al. (2003)) marks the control paths that lead to proof obligations; Jive (Meyer and Poetzsch-Heffter (2000)) allows the user to apply verification rules interactively; KIV (Haneberg et al. (2005)) and the Key tool (Ahrendt et al. (2007)) execute programs symbolically in dynamic logic. The problem that remains is that in proof obligations all approaches rely on backward-style reasoning, which goes against the operational understanding that programmers are familiar with (see Burdy et al. (2003)). Even symbolic execution of statement  $s$  in dynamic logic results in a proof obligation  $P \vdash \mathcal{U}Q$  where  $\mathcal{U}$  is an *update* that captures the effect of  $s$ . The update is then simplified by modifying the post-condition  $Q$  (Beckert, 2001, §5).

We propose to apply a guideline for designing user interfaces to the design of verification environments: The interface should consistently implement a metaphor for procedures that the user is already familiar with. One obvious candidate here is interactive debugging: Programmers usually have extensive experience with stepping through the source code, and examining the resulting states in detail to locate the source of an unexpected result. The corresponding metaphor for verification environments is this: The precondition of a Hoare triple  $\{P\}s\{Q\}$  captures the current state when statement  $s$  starts executing. The programmer should then be able to step through  $s$  and see how the precondition, and thus the “current state”, evolves with each executed step. The verification environment becomes a symbolic debugger that allows the programmer to inspect the behaviour of the program for all possible possible inputs.

We have implemented this approach in Isabelle/HOL for a low-level, C-like language  $L_0$ . The technical details of the development are given in (Gast (2008)). The focus there is on automated reasoning about disjointness of memory regions for simplifying memory updates. The purpose of this presentation is to demonstrate by examples the usefulness of the “verification as symbolic debugging” metaphor. Furthermore, we describe the specialized tactics that enable forward-style reasoning about programs in Isabelle, which is geared towards backward-style, goal directed proving.

## 2 The Hoare Logic for Forward Reasoning

The Hoare logic for  $L_0$  programs is designed to parallel the operational semantics of the language. A primary example is the rule for the assignment expression. Expressions can be evaluated as l- and r-values, as in C, using a standard big-step semantics (Gast, 2008, §3). The Hoare rules are then lemmata about relations  $\models \{P\} e \{Q\}$  and  $\models_r \{P\} e \{Q\}$  defined to capture partial correctness as usual. The assignment rule, in Isabelle’s syntax, is:

$$\begin{aligned} & \llbracket \models \{P_0\} e_1 \{P_1\}; \\ & \quad \forall a. (\models_r \{ \lambda \Gamma M. P_1 \Gamma M a \} e_2 \{ P_2 a \} \wedge \\ & \quad \quad (\forall \Gamma M v. P_2 a \Gamma M v \longrightarrow M \triangleright \text{typed-block } \Gamma a t \wedge \text{tyval } \Gamma v t)) \\ & \rrbracket \Longrightarrow \models \{P_0\} EAssign t e_1 e_2 \{ \lambda \Gamma M a. \exists M' v. P_2 a \Gamma (\text{STORE-TYPED } \Gamma a t M' M) v \wedge v = rd \Gamma a t M \} \end{aligned}$$

The context  $\Gamma$  maps local variables to their addresses,  $M$  and  $M'$  are memory states,  $a$  is an address, and  $v$  is a value. The assertions in the Hoare triples are (higher-order) predicates on these entities. The rule has an operational reading: If  $e_1$  takes a memory state described by  $P_0$  to a memory state (and result) described by  $P_1$ , and  $e_2$  takes that state to a state

described by P2, then after the assignment, P2 holds for the resulting memory state, except that reading from address  $a$  yields the value returned by  $e2$ , and a sequence of bytes has been modified, as represented by the STORE-TYPED operator (Gast (2008)). Since pointers are available, a side-condition checks that the accessed area is indeed allocated after  $e2$  has been executed. Note that the assertions P0, P1, and P2 inspect the passed memory state using the function  $rd$ . The generated post-condition will therefore contain, after  $\beta$ -reduction, sub-expressions  $rd \Gamma a' t'$  (STORE-TYPED  $\Gamma a t M' M$ ).

### 3 Forward Reasoning Tools

Most of Isabelle's tactics support backward-style, goal-oriented proving, while support for forward-style reasoning is very limited. To enable "verification as debugging", we have implemented the following specialized tactics.

**Unfolding and Folding of Assertions** A post-condition P generated by the forward application of Hoare rules has the form  $\lambda \Gamma M. \exists x_1 \dots x_n. P'$  where P' may contain any of the preceding bound variables. The user may manipulate assertions using weakening  $P \Rightarrow Q$ , where Q is a variable. Of the existing Isabelle tactics, only substitution and simplification can access subterms of P. We therefore *unfold*  $P \Rightarrow Q$  to a proper goal  $\bigwedge \Gamma M x_1 \dots x_n. P' \Longrightarrow Q$  on which all Isabelle tactics operate. In particular if P' is a conjunction, then all conjuncts will be available as separate assumptions in the goal. After the manipulation, the tactic `foldup` instantiates Q with a term  $\lambda \Gamma M. \exists x_1 \dots x_n. Q'$ .

**Lightweight Separation** The tactic `sep` (Gast (2008)) rewrites unfolded goals using conditional rules like the following: "is-valid (typed-block  $\Gamma a t \parallel$  typed-block  $\Gamma a' t'$ )  $\Longrightarrow$   $rd \Gamma a' t'$  (STORE-TYPED  $\Gamma a t M' M$ ) =  $rd \Gamma a' t' M$ "

The rule asserts that the memory access by  $rd$  is independent of the memory update represented by STORE-TYPED, if the accessed and modified address ranges do not overlap. `sep` uses assertions about the memory layout to prove this condition and drop memory update operator.

**Naming Old Values** Some accessor/modifier pairs will, of course, fail to simplify. If we have, for instance, an assertion about the content of a variable  $i$  and write to that variable, then the assertion contains a term  $rdv \Gamma i$  (STORE-VAR  $\Gamma i M' M$ ). The tactic `name_old_vals` replaces such terms with existentially quantified variables, in this case `old_i`.

**Factoring of Disjunctions** The rules for `if` statements and the short-circuit logical or operator generate post-conditions of the form  $\lambda \Gamma M. \exists x_1 \dots x_n. P_1 \Gamma M \vee \exists y_1 \dots y_m. P_2 \Gamma M$ , where  $P_1$  and  $P_2$  describe the post-state of the respective branches. Since both  $P_1$  and  $P_2$  have evolved from a common predecessor  $P_0$ , those conjuncts  $R$  of  $P_0$  that are not affected by the state updates in the branches will be duplicated. Also those existentially quantified variables  $x_1 \dots x_i$  that were already present in  $P_0$  will occur as  $y_1 \dots y_i$  as well. A tactic `factor_disj` replaces the above term with the simpler  $\lambda \Gamma M. \exists x_1 \dots x_i. R \wedge (x_{i+1} \dots x_n. P'_1 \Gamma M \vee \exists y_{i+1} \dots y_m. P'_2 \Gamma M)$ .

## References

- Wolfgang Ahrendt, Bernhard Beckert, Reiner Hähnle, Philipp Rümmer, and Peter H. Schmitt. Verifying object-oriented programs with KeY: A tutorial. In *5th International Symposium on Formal Methods for Components and Objects, Amsterdam, The Netherlands*, volume 4709 of LNCS, pages 70–101. Springer, 2007.
- Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Fourth International Symposium on Formal Methods for Components and Objects*, volume 4111 of LNCS. Springer, 2006.
- Bernhard Beckert. A dynamic logic for the formal verification of Java Card programs. In *JavaCard '00: Revised Papers from the First International Workshop on Java on Smart Cards: Programming and Security*, pages 6–24, London, UK, 2001. Springer-Verlag. ISBN 3-540-42167-X.
- Lilian Burdy, Antoine Requet, and Jean-Louis Lanet. Java applet correctness: A developer-oriented approach. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME*, volume 2805 of *Lecture Notes in Computer Science*, pages 422–439. Springer, 2003. ISBN 3-540-40828-2.
- Holger Gast. Lightweight separation. In *Theorem Proving in Higher Order Logics 21st International Conference, TPHOLS 2008*. Springer, 2008. (to appear).
- Dominik Haneberg, Simon Bumler, Michael Balsler, Holger Grandy, Frank Ortmeier, Wolfgang Reif, Gerhard Schellhorn, Jonathan Schmitt, and Kurt Stenzel. The user interface of the KIV verification system - a system description. In *Proceedings of the User Interfaces for Theorem Provers Workshop (UITP 2005)*, 2005.
- 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.