A Witnessing Compiler: A Proof of Concept

Kedar S. Namjoshi[1], Giacomo Tagliabue[2], and Lenore D. Zuck[2]

[1] Bell Laboratories, Alcatel-Lucent
kedar@research.bell-labs.com
[2] University of Illinois at Chicago
giacomo.tag@gmail.com, lenore@cs.uic.edu

Abstract. In prior work we proposed a mechanism of "witness generation and propagation" to construct proofs of the correctness of program transformations. Here we present a simpler theory, and describe our experience with an initial implementation based on the LLVM open-source compiler and the Z3 SMT solver.

1 Introduction

Ensuring the correctness of an optimizing compiler is a classic question in computing. Compilers are very large programs – for instance, GCC is over 7 million lines of code, and LLVM is near a million – and they carry out the essential task of transforming other programs (often themselves large) into executable machine code. Ensuring the correctness of compiler transformations is thus a critical question; however, manual inspection is impossible, which has led to much work on the construction of automated proofs of correctness.

In [7] we proposed a methodology for creating such a proof. There, each optimization procedure in the compiler is augmented with an auxiliary *witness generator*. For every instance of optimization, the generator constructs, *at run time*, a *witness relation* between its target and source programs. The conditions for a relation to be a proper witness are checked off-line, using an automated theorem prover. Thus, when a witnessing compiler is used on a source program, it generates a chain of witnesses, one for each optimization, which connects the source program with the final target program. Each link of the chain may be verified *independently* of the compiler code.

Witness generation can be positioned in-between two well known methods of compiler verification: machine-checked proofs of correctness (e.g., [5]) and Translation Validation (TV) (e.g., [10]). It is substantially simpler to define a witness generation procedure than to prove an optimization correct, as the definition does not require one to show (or assume) the correctness of the analysis phase of an optimization. Moreover, as the generating procedure is written with full knowledge of the optimization, one avoids the heuristic constructions which limit the scope of translation validation. The potential drawback to witness generation is the run-time overhead of generation and checking.

In this paper, we report on early experiments with witness generation. The implementation is carried out using the LLVM compiler framework [4]. It currently supports a limited set of instructions (enough to represent **while** programs over the integers) and a small set of transformations (simple constant propagation, dead-code-elimination, loop invariant code motion). The generated witnesses are checked for validity with the Z3 SMT solver [3]. Our experience has been encouraging: the witness generation approach is feasible and requires only small amounts of additional code. The overhead of witness checking is high, but we expect this to reduce with better implementation techniques.

2 Transformations and Witnesses

This section summarizes ideas described in more depth in [7].

Definition 1 (Program). *A program is described as a tuple* $(V, \Theta, \mathcal{T})$*, where*

- V *is a finite set of (typed)* state variables*, including a distinguished* program location variable*,* π*,*
- Θ *is an* initial condition *characterizing the initial states of the program,*
- $\mathcal{T}$ *is a* transition relation*, relating a state to its possible successors.*

A program *state* is a type-consistent interpretation of its variables. The transition relation is denoted syntactically as a predicate on V and V', which is a primed copy of V. For every variable x in V, its primed version x' refers to the value of x in the successor state.

To match the LLVM structure, we consider programs described by a control flow graph (CFG), where each node is a *basic block* (BB) consisting of a single-entry single-exit straight line code. The transition relation of the program can thus be viewed as a disjunction of transition relations $\mathcal{T}_{ij}$, each describing the transition between basic block i (BB_i) and basic block j (BB_j) such that BB_j is an immediate successor of BB_i. The program location variable π ranges over the set of basic block identifiers. We assume that a CFG has a unique *initial* BB with no incoming edges, and a unique *terminal* BB without outgoing edges.

A *witness* relation connects the values of source and target program locations at corresponding basic blocks. In the simplified view, we define a witness relation to have two components:

- A *control mapping* κ from the basic blocks of T to those of S. The function κ maps the initial block of T to the initial block of S, and the terminal block of T to that of S.
- A *data relation*, $\varphi_{i,\kappa(i)}(V_T, V_S)$, which connects the values of target and source variables at corresponding blocks i and $\kappa(i)$. For this paper, it suffices to have relations which are defined as conjunctions of the form $v = e$ where v is a program variable and e is an expression, over variables of either S or T. For instance, one can define equality of corresponding source and target variables by a set of conjunctions of this form.

There are three conditions, shown in Figure 1, which are checked to ensure that a relation is a proper witness (i.e., it ensures the correctness of the transformation). The first checks that the witness relation is a (stuttering) simulation;

the second, that source and target variables match at initial and final blocks. The stuttering simulation check allows infinite stuttering on the source program side; this can be fixed, as described in [7], by generating an auxiliary ranking function. As our current implementation does not do that, we omit it from the rule. The predicate $oeq(V_T, V_S)$ (read as "observably equal") asserts that corresponding target and source variables are equal in value. The correspondence is specific to the optimization. (Hence, a witness is correct up to a correspondence.)

1. For every target block i, the following implication must be valid. $[\varphi_{i,\kappa(i)}(V_T, V_S) \wedge \mathcal{T}^T_{ij}(V_T, V'_T) \Rightarrow (\exists V'_S : (\mathcal{T}^S_{\kappa(i),\kappa(j)}(V_S, V'_S) \wedge \varphi_{j,\kappa(j)}(V'_T, V'_S))) \vee \varphi_{j,\kappa(i)}(V'_T, V_S)]$
2. For the initial block a, $[(\exists V_S : \varphi_{a,\kappa(a)}(V_T, V_S) \wedge oeq(V_T, V_S))]$ must be a validity.
3. For the final block f, $[\varphi_{f,\kappa(f)}(V_T, V_S) \Rightarrow oeq(V_T, V_S)]$ must be a validity.

Fig. 1. Witness Checking

Typically, a witness relation encodes invariants about the source and target programs, which are inferred during the analysis phase of an optimization. For instance, constant propagation generates assertions about which variables of the source program are constant, and dead-code elimination depends on a liveness analysis that generates assertions about the live variables at each program point. The witness relation for constant propagation, for example (see [7]), states that $(x_T = x_S)$ for every variable x and that $(x_S = c)$ for those variables x which are known to have constant value c at the source location $\kappa(i)$.

3 Implementation

The source code of the implementation is a fork from LLVM, and is available as a git repository at `https://bitbucket.org/itajaja/llvm-csfv`. Currently, the implementation targets the intra-procedural optimization passes in LLVM, defined over its intermediate representation (IR). Programs in the IR are in SSA (single static assignment) form for each function.

The process that is followed to build a witnessing pass is similar for every pass. The starting base is the LLVM source code for an optimization pass. First, the analysis phase of the pass is augmented – if needed – to store all the invariants found by the analysis for each program location (or basic block). These invariants are used for the witness generation. To validate a witness, it is necessary to build the transition relations for the source and target programs. The validation checks implement the proof rule in Fig. 1 using the Z3 SMT solver. As basic blocks are (guarded) deterministic code fragments, the existential quantification in the simulation check can be eliminated, which simplifies the check.

Table 1. Measurements

Pass	Original LOC	Witness Gen. LOC	Avg. runtime in ms (overhead multiple)
Simple Constant Propagation	99	118	101.36 (12x)
Dead Code Elimination	135	37	41.71 (10x)
Loop Invariant Code Motion	895	65	200.03 (31x)

The framework design is based on the following main components: Optimizer/Analyzer, Witness Generator, Translator, Witness Checker, Invariant Propagator. The *Optimizer/Analyzer* augments the LLVM pass to store the analysis invariants; the *Witness Generator* takes care of generating the optimization-specific witness using the invariants found during the analysis; the *Translator* builds the transition relation of a given CFG and is usually run over the target and the source of every optimization pass; and the *Witness Checker* combines the generated witness and the target and source transition relation to verify that the witness is a stuttering simulation using Z3. In addition, an *Invariant Propagator* uses the witness relation and symbolic manipulations using Z3 to propagate invariants (computed during analysis or externally supplied) from a source program to the target. Out of these five components only the first two are optimization-specific.

Table 1 gives measurements which show (a) the effort required to write a witness generator and (b) the overhead incurred to check the correctness of witnesses. The implemented passes are chosen by their commonality, ease of study and for clearly highlighting some of the critical parts of the framework. The lines of code (LOC) for witness generation are those that are required specifically for that optimization. In addition, there is code which is common to all passes, and implements a witness checker, the invariant propagator, the translator, and basic definitions, amounting in total to approximately 1 KLOC.

The LOC numbers are encouraging: compared to the effort required to define the optimization, the effort required to define a witness generator is high only for the simple constant propagation pass, but is much lower for the other two passes. The run-time overhead measures the overhead of witness generation and checking compared to the optimization time, measured with the `time-passes` tool of the LLVM optimizer. The current runtime overhead for witness checking is very high. However, this is a rough, unoptimized implementation, so we expect this overhead to reduce substantially as the implementation is improved.

4 Conclusion and Related Work

The implementation described here is a work in progress, and is currently at an early stage. Support for the instruction set of the IR is limited to binary operations over integers, return, branch (conditional and unconditional), compare, and ϕ nodes. (This set suffices to describe **while** programs over the integers.) For this reason, it is not possible yet to test the framework against "real" programs that contain many currently unsupported instructions and data types.

Ensuring the correctness of program transformations – in particular, compiler optimizations – is a long-standing research problem. In [6], Leroy gives a nice technical and historical view of approaches to this question. A primary approach is to formally prove each transformation correct, over all legal input programs. This is done, for example, in the CompCert project [5], and in [2], which derives and proves correct optimizations using denotational semantics and a relational version of Hoare's logic Formal verification of a full-fledged optimizing compiler is often infeasible, due to its size, evolution over time, and, possibly, proprietary considerations. *Translation Validation* offers an alternative to full verification. A primary assumption of this approach is that the validator has limited knowledge of the transformation process. Hence, a variety of methods for translation validation arise (cf. [9,8,11,13,14,12]), each making choices between the flexibility of the program syntax and the set of possible optimizations that are handled. As details of the optimization are assumed to be unknown, heuristics are used, which naturally limits the scope of the method. Recently, [1] proposes a method for proving equivalence based on relational Hoare logic; it resembles our witnesses, yet is closer to translation validation and has similar limitations.

Since we assume the optimization process is visible to the witness generator, the generator is able to make use of auxiliary invariants derived by the optimizer in order to produce a witness. This implies that witness generation is, in principle, applicable to any optimization.

Acknowledgements. This material is based on research sponsored by DARPA under agreement number FA8750-12-C-0166. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

References

1. Barthe, G., Crespo, J.M., Kunz, C.: Beyond 2-safety: Asymmetric product programs for relational program verification. In: Artemov, S., Nerode, A. (eds.) LFCS 2013. LNCS, vol. 7734, pp. 29–43. Springer, Heidelberg (2013)
2. Benton, N.: Simple relational correctness proofs for static analyses and program transformations. In: POPL, pp. 14–25 (2004)
3. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
4. Lattner, C., Adve, V.S.: LLVM: A compilation framework for lifelong program analysis & transformation. In: CGO, pp. 75–88 (2004), `llvm.org`
5. Leroy, X.: Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In: POPL, pp. 42–54. ACM (2006)

6. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM 52(7), 107–115 (2009)
7. Namjoshi, K.S., Zuck, L.D.: Witnessing program transformations. In: Logozzo, F., Fähndrich, M. (eds.) SAS 2013. LNCS, vol. 7935, pp. 304–323. Springer, Heidelberg (2013)
8. Necula, G.: Translation validation of an optimizing compiler. In: Proceedings of the ACM SIGPLAN Conference on Principles of Programming Languages Design and Implementation, PLDI 2000, pp. 83–95 (2000)
9. Pnueli, A., Siegel, M., Shtrichman, O.: The code validation tool (CVT)- automatic verification of a compilation process. Software Tools for Technology Transfer 2(2), 192–201 (1998)
10. Pnueli, A., Siegel, M., Singerman, E.: Translation validation. In: Steffen, B. (ed.) TACAS 1998. LNCS, vol. 1384, pp. 151–166. Springer, Heidelberg (1998)
11. Rinard, M., Marinov, D.: Credible compilation with pointers. In: Proceedings of the Run-Time Result Verification Workshop (July 2000)
12. Tristan, J.-B., Govereau, P., Morrisett, G.: Evaluating value-graph translation validation for LLVM. In: PLDI, pp. 295–305 (2011)
13. Zuck, L.D., Pnueli, A., Goldberg, B.: Voc: A methodology for the translation validation of optimizing compilers. J. UCS 9(3), 223–247 (2003)
14. Zuck, L.D., Pnueli, A., Goldberg, B., Barrett, C.W., Fang, Y., Hu, Y.: Translation and run-time validation of loop transformations. Formal Methods in System Design 27(3), 335–360 (2005)