Witnessing An SSA Transformation

Kedar S. Namjoshi
Bell Labs, Alcatel-Lucent
kedar@research.bell-labs.com

ABSTRACT

The correctness of program compilation is important to assuring the correctness of an overall software system. In this work, we describe an effort to verify the compiler transformation which turns memory references into corresponding register references. It is one of the most important transformations in modern compilers, which rely heavily on the SSA (static single assignment) form produced by this transformation. Formally verifying the algorithm behind this transformation is thought to be a difficult task. Verifying the actual code, as implemented in a production compiler, is currently infeasible. We describe our experience with an alternative verification strategy, which is based on generating and checking "witnesses" for each instance of the transformation. This approach enormously simplifies the verification task, primarily because it does not require showing that the transformation code is correct.

1. INTRODUCTION

In modern compilers, optimizations are carried out on programs in the SSA (static single assignment) format. It is usually easier for the front-end of a compiler to produce code that is trivially in SSA form, which must be transformed to replace memory references (i.e., load, store) with corresponding register references (i.e., read, write) before further optimizations can be applied. In the LLVM compiler [5], this crucial transformation is called `mem2reg`, and it can account for more than half of the optimization speedup on typical programs, as shown in [17]. In that paper, which is primarily concerned with verification, the authors suggest that formally verifying the algorithms behind `mem2reg` (from [4, 16]) would be difficult, as the algorithms break SSA constraints at intermediate stages. (The authors define and verify their own algorithm for memory-to-register transformation.) Verifying the actual implementation of `mem2reg` is currently infeasible: it would require a formalization of C++ semantics and proofs of correctness of much more than the approximately 1500 lines of C++ code which implements `mem2reg`. In a nutshell, formally establishing the correctness of the `mem2reg` implementation is an open question and a challenge for proof methods.

In this paper, we report on our experience with an alternative verification strategy, which is based on generating and checking "witnesses" for the correctness of each instance of a transformation [10]. I.e., this strategy proves the correctness of `mem2reg` on a single program at a time; a conventional proof would show correctness over all programs at once. We add auxiliary code to the `mem2reg` implementation, so that when it is invoked on a program S, the code emits hints as to why the transformation from S to the result program, T, is correct. The hints are gathered and processed into a logical refinement relation which links the state space of T with that of S. This candidate refinement relation is checked using a general-purpose refinement checker. If it is correct, we have shown correctness for this instance of the transformation, and have a concrete "witness" to justify that claim.

In our experience, this approach enormously simplifies the human effort required to show correctness. We need only about 160 lines of auxiliary C++ code to produce hints, and 500 lines of OCaml code to process the hints into a logical refinement relation. Contrast this with the proof described in [17] (for a different algorithm, as noted earlier), which is based on about 10,000 lines of Coq proof script. The refinement checker is about 850 lines of OCaml code, and relies on SMT solvers to carry out the validity checks. The checker can, of course, be used to check witnesses produced by other optimizations. The flip side of our approach is that it may require a significant amount of computation to check a witness relation. In essence, one substitutes a huge, one-time, human effort with a recurring computational effort.

Hint and witness generation requires a good understanding of the `mem2reg` implementation as well as the underlying algorithm. The LLVM implementation does not directly follow the original algorithms: it handles several special cases separately before applying an optimized form of the original algorithm. In this short paper, we sketch the process of witness generation, describing the hints that arise, and how they are processed into a logical refinement relation. The description is deliberately kept informal, in order to give a feel for the process to be followed in writing a witness generator. We conclude with some initial experimental results.

2. WITNESS GENERATION

An example of the general transformation is given in Figure 2. (The program syntax in this and other figures is a simplification of the LLVM intermediate format, omitting types and other attributes. Our implementation handles the full LLVM syntax.) For this example, the transformation replaces loads and stores on the allocated memory represented by `x` with reads and writes to the newly defined register variable `x.0`. The "phi" function in the `while.cond` block is a device added during the transformation to ensure that `x.0` obtains the correct value based on control flow: if block `while.cond` is reached from block `entry`, this value is 0; otherwise, it is the value in register `add`.

In its essence, `mem2reg` is a renaming of memory locations to registers. The memory locations to which the transformation is applied hold addresses of dynamically allocated stack memory. In the intermediate representation, such a location is identified by the target `x` of an instruction `x = alloca`. The SSA constraint implies that the renaming is, in general, a one-to-many relation: i.e., a single memory location in the source may be mapped to different register names at different points in the target program. The control-flow structure of the source program is unchanged.

As `mem2reg` is a renaming, we expect its refinement map to be a conjunction of assertions of the form "the value at memory address x in the source program equals the value of register r in the target program". More fully, the map is given by a set of relations, $W_e(C_T, C_S)$, where e is an edge of the control flow graph, S is the source program, and T is the result of transformation. The notation $C_S = (M_S, R_S)$ represents the configuration (or state) of program S: its memory map is given by M_S and its register map by R_S. Precisely which (x, r) relationships are included in W_e is determined by the transformation. The process of witness generation is one of extracting those relationships from the algorithm and the implementation code. To illustrate this process, we consider one of the special cases of the transformation, as well as the general case. A full technical report with an account of all cases is under preparation.

2.1 Special Case: single store

The special case we consider is one where there is only a single store to an allocated memory location (in LLVM, the `RewriteSingleStoreAlloca` pass), illustrated by the example in Figure 1. Notice that the transformation only deletes loads and stores to `x` from the source program, and removes the registers which hold the values obtained from loads. No registers are added to the target program.

There are two important aspects to the transformation.

- A register that is the target of a load from `x` is removed and every use of that register is replaced with the (symbolic) value that it would have obtained from the load. For instance, `t6` in block `b2` is removed, and its use in block `bfinal` is replaced with the value it should have, which is `p`.

  In order to ensure that block `bfinal` in the target is a refinement of block `bfinal` in the source, it is necessary to have the refinement assertion $R_S[\texttt{t6}] = R_S[\texttt{p}]$ associated with the edge $(\texttt{b2}, \texttt{bfinal})$. The witness generator produces the hint that `t6` is replaced with `p` at `b2`, and that this information should be propagated to all CFG edges which are dominated by `b2`, i.e., the edge $(\texttt{b2}, \texttt{bfinal})$.

- The value stored into the alloca is also propagated down the control flow graph. For instance, one cannot show that `b0` in the target is a proper refinement of `b0` in the source without knowing that the value that will be loaded from `x` in the source block is `p`. Hence, the witness generator produces the hint $M_S[R_S[\texttt{x}]] = R_S[\texttt{p}]$, which says that `p` is the value given to memory location `x` in block `init`, and that that this information should be propagated to all CFG edges which are dominated by `init`, in particular, the edge $(\texttt{init}, \texttt{b0})$.

Notice that the hints constrain only the source program state, which is also true for the other special cases.

2.2 The general transformation

If none of the special cases apply, one obtains the general case where new registers are introduced in the target. This is called the `RenamePass` case in LLVM. An example is shown in Figure 2. The algorithm from [16] is used (with improvements) to determine the minimal number of phi-nodes and registers that are necessary. The `RenamePass` procedure then fills in the new phi-node entries with a dataflow calculation: for each block and source address x, the values corresponding to x flow in to a phi-node from all previous nodes and a new value for x flows out to all successor nodes.

The witness generator tracks this flow of values. For the example, the value 0 flows into the phi-node from the edge $(\texttt{entry}, \texttt{while.cond})$. For the refinement relation to hold, it is necessary that the value of `x` is known to be 0 along this edge in the source (otherwise, the value loaded from `x` in block `while.cond` of the source program is unconstrained). The generator produces the assertion $M_S[R_s[\texttt{x}]] = 0$ for this edge. Similarly, it produces the assertion $M_S[R_S[\texttt{x}]] = R_T[\texttt{add}]$ for the other edge, $(\texttt{while.body}, \texttt{while.cond})$. The dataflow propagation ensures that `x.0` is the value associated with `x` on exit from `while.cond`, so the generator produces the assertion $M_S[R_S[\texttt{x}]] = R_T[\texttt{x.0}]$ for the edge $(\texttt{while.cond}, \texttt{while.body})$.

2.3 Constructing the full refinement relation

The hints described previously are not enough in themselves to show refinement; we must augment them with the assertions below.

- (memory equality) $(M_S =_A M_T)$. In LLVM, the memory locations that are turned into SSA registers are called "promotable allocas". The notation $=_A$ says that the source and target program memories are identical except at the addresses of the promoted allocas.
- (register equality) $R_S[r] = R_T[r]$, for all registers r which are common to the source and target.

```
int foo(int p) {
   init:
       x = alloca;
       store p x;
       cmp = lt p 1;
       br cmp b0 b1;
   b0:
       t2 = p+1;
       t3 = load x;
       t4 = t2 + t3;
       br bfinal;
   b1:
       t5 = p+2;
       br b2;
   b2:
       t6 = load x;
       br bfinal;
   bfinal:
       t7 = phi (b0,t4) (b2,t6);
       return t7;
```

```
int foo(int p) {
   init:
       cmp = lt p 1;
       br cmp b0 b1;
   b0:
       t2 = p+1;
       t4 = t2 + p;
       br bfinal;
   b1:
       t5 = p+2;
       br b2;
   b2:
       br bfinal;
   bfinal:
       t7 = phi (b0,t4) (b2,p);
       return t7;
}
```

Figure 1: Special Case: Single Store. Source on left, target on right.

```
function foo(){
  entry:
    x = alloca;
    store  0 x;
    branch while.cond;
  while.cond:
    tmp = load  x;
    cmp = le tmp 100;
    branch cmp while.end while.body;
  while.body:
    tmp1 = load  x;
    add =  add tmp1 1;
    store  add x;
    branch while.cond;
  while.end:
    tmp2 = load  x;
    return  tmp2;
}
```

```
function foo(){
  entry:
    branch while.cond;
  while.cond:
    x.0 = phi (entry 0) (while.body add);
    cmp = le x.0 100;
    branch cmp while.end while.body;
  while.body:
    add = add x.0 1;
    branch while.cond;
  while.end:
    return  x.0;
}
```

Figure 2: General Transformation. Source on left, target on right. The example is from [17].

2.4 Witness checking

In our implementation, the hints are generated by auxiliary C++ code added to the LLVM `mem2reg` implementation. The propagation of hints and the construction of the full refinement relation is carried out by a separate procedure, programmed in OCaml. This procedure reads in the source and target programs, constructs the dominance relation for the source CFG, and uses this information to propagate hints. The separation of work between languages is mostly a matter of programming convenience: one could also use the dominance relations computed by LLVM to handle propagation.

Witness checking is carried out by a general refinement checker which has no knowledge of `mem2reg`. This checker receives as input the source (S) and target (T) programs, and a witness relation (W). The witness relation is (conceptually) specified as a collection of triples (e, f, w) where e is a target CFG edge, f is a source CFG edge, and w is a relation between the configurations of S and T. (For `mem2reg`, it is the case that $e = f$ for all witness triples.)

The refinement check is essentially a check that the witness relation is a simulation relation from T to S. It operates as follows. For a witness triple (e, f, w), where $e = (u, v)$ and $f = (m, n)$:

1. For each transition from block v to v', the check determines a successor block n' of n and ensures that the witness w' associated with the new pair of edges, $e' = (v, v')$ and $f' = (n, n')$, holds after the transitions of blocks v and n. I.e., letting $\mathsf{btrans}(b)$ be the transition relation of block b, the following implication should be valid:

   $$[w(C_T, C_S) \wedge \mathsf{btrans}(v)(C_T, C'_T) \wedge \mathsf{btrans}(n)(C_S, C'_S) \Rightarrow w'(C'_T, C'_S)]$$

   This is a simplification of the standard simulation condition assuming that btrans is deterministic, which eliminates the existential quantifier on C'_S. In general, one may also need to add auxiliary history variables and allow stuttering [10].

2. If v' is an exit node, the check also ensures that the return values from v and n are identical.

The simulation check is encoded in SMT in a rather straightforward manner. M and R are defined as $\mathsf{Int} \to \mathsf{Int}$ arrays;

register names are distinct integer constants; and all operators are uninterpreted. This default suffices for `mem2reg`; for other transformations such as constant propagation, one must interpret some of the operators.

2.5 Early Experiments

The prototype implementation has been tested on several small-to-medium size C programs. Two representative examples are the GNU library `microhttpd` (about 8K lines of C) and the model checker `SPIN` (about 20K lines of C). For `microhttpd`, the witness checker issues 4603 queries which take 1760 seconds. Some queries (114, or 2.4%) do not succeed: they either time out (the limit is set to 5 seconds per query) or are invalid. The timeout cases we have examined are all valid. The invalid cases appear to be corner-case situations where incomplete witness relations are created, rather than representing failures of the `mem2reg` implementation. For `SPIN`, the checker issues 21,908 queries which take 3.5 hours to discharge. The number of incomplete checks is small (146, or 0.66%). A couple of files could not be processed; they contain variants of LLVM instructions which are not currently handled by the checker. The checker is a work in progress, so we expect to make improvements in speed as well as in overall accuracy.

3. RELATED WORK

Full correctness proofs of compiler transformation (e.g., [8, 3, 6, 17] and related work) are complex. The primary source of the difficulty is that one has to formulate the right invariants which which to connect the state of the transformation code to the state and the semantics of the (unspecified) program that is being transformed. As an example, the proof of a dead-code-elimination transformation must include a sub-proof which shows the correctness of the fixpoint algorithm used to compute liveness information. The witnessing approach avoids such proofs: one need not know whether the live variable computation is correct for all programs, it suffices to check whether its results for the given program produce the expected refinement relation. In the `mem2reg` analysis, one similarly avoids the need to show that the computation of phi-node placement is correct. Neither the transformation code nor the witness generator are required to be correct.

This generate-and-check approach to correctness builds on and has strong connections to Proof-carrying Code [12], Credible Compilation [14], and Translation Validation [13, 11]. We discuss each in turn. Proof-carrying code is a way of ensuring that a single program satisfies a correctness property. As we consider program transformations, the proof is not fixed, but rather is created by a proof generator for each input program. The witnessing approach is a form of translation validation in the broad sense. However, existing methods for translation validation use heuristics to (implicitly) compute a refinement relation between programs. We believe that the witness generation method is a better and simpler option when the code of the translator is available and can be modified. Credible Compilation and witness generation are, conceptually, broadly similar, but differ in technical detail – credible compilation uses a restricted refinement relation which does not allow for stuttering or history. Implementations of credible computation (cf. [15, 7]) have not tackled transformations of the complexity of `mem2reg`. In [2, 1], the authors describe a related approach to verifying memory-to-register transformation. They provide a verified checker for the CompCert compiler [6] which (in our notation), given S and T, decides whether T is a valid SSA-form of S. A significant difference in the approach described here is that our (trusted) checker does a general refinement check: the presence of the witness relation allows the checker to be written in a form that is not specialized for the memory-to-register transformation. In [9], we described a much earlier version of the checker, which handled only a small subset of the LLVM syntax.

Compiler transformations appear to be in a "sweet spot" for a generate-and-check approach to verification. While a compiler optimization algorithm and its implementation can be quite complicated, the effect of the transformation (i.e., the refinement relation) is, in most cases, easy to describe with a combination of equality, uninterpreted functions, arithmetic and arrays – which happen to be theories that are well supported by SMT solvers.

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.

I would like to thank my collaborators on the broader project of constructing self-certifying compilers for encouragement and many helpful comments during the course of this work: Lenore Zuck, V. N. Venkatakrishnan, Jens Palsberg, Liana Hadarean, Tim King, and Oswaldo Olivo.

4. REFERENCES

[1] G. Barthe, D. Demange, and D. Pichardie. A Formally Verified SSA-Based Middle-End - Static Single Assignment Meets CompCert. In H. Seidl, editor, *ESOP*, volume 7211 of *Lecture Notes in Computer Science*, pages 47–66. Springer, 2012.

[2] G. Barthe, D. Demange, and D. Pichardie. Formal Verification of an SSA-Based Middle-End for CompCert. *ACM Trans. Program. Lang. Syst.*, 36(1):4, 2014.

[3] W. R. Bevier, W. A. Hunt, J. S. Moore, and W. D. Young. An approach to systems verification. *J. Autom. Reasoning*, 5(4):411–428, 1989.

[4] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991.

[5] C. Lattner and V. S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88, 2004. Webpage at llvm.org.

[6] X. Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In

POPL, pages 42–54. ACM, 2006.

[7] D. Marinov. Credible compilation. Master's thesis, MIT, 2000.

[8] J. McCarthy and J. Painter. Correctness of a compiler for arithmetic expressions. pages 33–41. American Mathematical Society, 1967.

[9] K. S. Namjoshi, G. Tagliabue, and L. D. Zuck. A witnessing compiler: A proof of concept. In A. Legay and S. Bensalem, editors, *RV*, volume 8174 of *Lecture Notes in Computer Science*, pages 340–345. Springer, 2013.

[10] K. S. Namjoshi and L. D. Zuck. Witnessing program transformations. In F. Logozzo and M. Fähndrich, editors, *SAS*, volume 7935 of *Lecture Notes in Computer Science*, pages 304–323. Springer, 2013.

[11] G. Necula. Translation validation of an optimizing compiler. In *Proceedings of the ACM SIGPLAN Conference on Principles of Programming Languages Design and Implementation (PLDI) 2000*, pages 83–95, 2000.

[12] G. Necula and P. Lee. Safe kernel extensions without run-time checking. In *OSDI*, 1996.

[13] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In B. Steffen, editor, *TACAS*, volume 1384 of *Lecture Notes in Computer Science*, pages 151–166. Springer, 1998.

[14] M. Rinard. Credible compilation. Technical Report MIT-LCS-TR-776, MIT, 1999.

[15] M. Rinard and D. Marinov. Credible compilation with pointers. In *Proceedings of the Run-Time Result Verification Workshop*, July 2000.

[16] V. C. Sreedhar and G. R. Gao. A linear time algorithm for placing phi-nodes. In R. K. Cytron and P. Lee, editors, *POPL*, pages 62–73. ACM Press, 1995.

[17] J. Zhao, S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. Formal verification of SSA-based optimizations for LLVM. In H.-J. Boehm and C. Flanagan, editors, *PLDI*, pages 175–186. ACM, 2013.