Translation Validation: Automatically Proving the Correctness of Translations Involving Optimized Code

Hanan Samet

http://www.cs.umd.edu/~hjs

hjs@cs.umd.edu

Department of Computer Science
University of Maryland
College Park, MD 20742, USA

Compiler Testing (Now Known as Translation Validation and Proof-Carrying Code)

Definition: a means for proving for a given compiler (or any program translation procedure) for a high level language H and a low level language L that a program written in H is successfully translated to L.

Motivation is desire to prove that optimizations performed during the translation process are correct:
1. Often, optimizations are heuristics
2. Optimizations could be performed by simply peering over the code

Proof procedure should be independent of the translation process (e.g., compiler).

Notion of correctness must be defined carefully.

Need a representation that reflects properties of both the high and low level language programs. Must identify:
1. Critical semantic properties of high level language
2. Interrelationship to instruction set of computer executing the resulting translation
Prior Work in Program Verification

- We are interested in proving that programs are correctly translated.
- Different from proving that programs are correct.
- Different from showing that program is correct for a given input(s).
- Historically, attempts have been based on use of assertions about the intent of the program which are then proved to hold (Floyd, King).

Difficulties include:

2. How to allow for possibility that assertions are inadequate to specify all the effects of the program in question.

No need for any knowledge about purpose of program to be translated:

1. Many possible algorithms for sorting (e.g., Quicksort, Shellsort, Insertion Sort, etc.)
2. To prove equivalence of any two of these algorithms, we must demonstrate that they have identical input/output pairs.
3. Conventional proof systems attempt to show that the algorithms yield identical results for all possible inputs.
4. Proving equivalence of different algorithms is known to be generally impossible by use of halting problem-like arguments.
Our Approach

- In order to avoid unsolvability problem, need to be more precise on the definition of equivalence

- By equivalence we mean that two programs must be capable of being proved to be structurally equivalent (termed “syntactic correspondence”)
  - Alternatively, must have identical execution sequences
  - Must test same conditions except for certain valid rearrangements of computations

- We prove correctness of the translation

Current realizations and efforts:

- Originated as Compiler Testing by Samet in Ph.D. thesis in 1975
- Certifying Compiler or Proof-Carrying Code by Necula and Lee in 1996
- Rediscovered by Pnueli, Siegel, and Singerman in 1998 and termed it Translation Validation and followed by Zuck, Pnueli, Fang, and Goldberg in 2003
- Acknowledgment of relationship to Samet’s work includes Blech, Buttle, Gawkowski, Gregoire, Jourdan, Kundu, Leinenbach, Lerner, Leroy, Pottier, Rideau, Shashidhar, Stepp, Stringer-Calvert, Tate, Tatlock, Tristan, and Zimmerman
Alternative Approaches

- One method is to prove that there does not exist a program which is incorrectly translated by the compiler.

- Instead, we prove that for each program input to the translation process, the translated version is equivalent to the original version:
  1. A proof must be generated for each input to the translation process.
  2. Advantage is that as long as the compiler performs its job for each program input to it, its correctness is of a secondary nature.
  3. Proof system can run as a postprocessing step to compilation.
  4. We have bootstrapped ourselves so that we can attribute an “effective correctness to the compiler.”
  5. The proof process is independent of the compiler and thus proof system also holds for other compilers from the same source and target languages as well as some manual translations and optimizations.
  6. Identifies proof as belonging to the semantics of the high and low level languages of the input and output rather than the translation process.

- A method that would prove a particular compiler correct is limited with respect to the types of optimizations that it could handle as it would rely on the identification of all possible optimizations a priori (e.g., LCOM0 and LCOM4 of McCarthy).
# Historical Perspective I

1. **McCarthy and Painter** 1967: Proved correctness of an arithmetic expression compiling algorithm

2. **Milner** 1971: Proposed “simulation” as a way to capture fact that two programs realize the same algorithm but did not apply to compiler output

3. **Kreisel** 1971: Discusses notion of “checking” of programs and calls for checking equivalence of programs through normalization transformations

4. **Milner and Weyhrauch** 1972: Present a machine-checked proof (using LCF) of the correctness of McCarthy and Painter’s compiling algorithm

5. **Samet** 1972-1975: Proposed proving correctness of compilers by showing source and target are equivalent and independent of the compiler
   - Notion of equivalence of programs (termed “syntactic correspondence”) similar to Milner’s notion of “simulation” of programs

6. **Blum and Kannan** 1989: Distinguish between verification and testing of programs and proposed checking of programs as being in between
   - Checking: verifying that program returns a correct answer for each input given to it rather than for all inputs
   - Samet 1972-1975: Originated application of checking to a compiler and that the check is independent of the compiler
   - No mention or reference to the work of Samet
Historical Perspective II

Necula and Lee 1996 and Pnueli et al. 1998: certifying compiler

1. Necula and Lee 1996: Proof-carrying code: proof is part of certifying compiler
   - Same as Samet when embed Samet’s check in the compiler
   - No mention or reference to the work of Samet

2. Pnueli et al. 1998: Translation validation: decouple compiler and proof
   - Same as Samet where proof is independent of compiler
   - No mention or reference to the work of Samet

Recent work acknowledging contributions of Samet


5. US: Kundu Tatlock, and Lerner (2009); Tate, Stepp, Tatlock, and Lerner (2009, 2011); Tatlock and Lerner (2010); Stepp, Tate, and Lerner (2011)
Equivalence proof applies equivalence preserving transformations in an attempt to reduce them to a common representation termed a normal form.

Symbolic interpretation is different from:

1. Symbolic execution where various cases of a high level language program are tested by use of symbolic values for the parameters
2. Decompilation as don’t return source high level program
Example

- High level language: LISP 1.6
- Low level language: LAP (variant of DECsystem-10 assembly language)
- Example function: intersection of two lists U,V

```lisp
procedure INTERSECTION(U,V)
    1 if NULL(U) then NIL
    2 elseif MEMBER(CAR(U),V) then
    3     CONS(CAR(U),INTERSECTION(CDR(U),V))
    4 else INTERSECTION(CDR(U),V)
    5 endif

Sample input/output: INTERSECTION('(A B C), '(D C B)) = '(B C)
```
Flowchart of Conventional LAP Encoding

ENTER: R1=U
R2=V
STACK <= R1
STACK <= R2

EQ(U, NIL)?

YES
R1 = CAR(R1)
R1 = MEMBER(R1, R2)

NO
MEMBER( CAR(U), V)?

NO
R1 = CAR(STACK(-1))
R2 = STACK(0)
STACK <= R1
R1 = CDR(STACK(-1))
R1 = INTERSECTION(R1, R2)
STACK >= R2
R1 = XCONS(R1, R2)

YES
R2 = STACK(0)
R1 = CDR(STACK(-1))
R1 = INTERSECTION(R1, R2)

undo the first two stack operations
RETURN(R1)

END
# Example Optimized LAP Encoding

- Obtained by hand optimization process

<table>
<thead>
<tr>
<th>Interpreted Code</th>
<th>Optimized Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTERSECTION</td>
<td>JUMPE 1 TAG 1</td>
</tr>
<tr>
<td></td>
<td>JUMP TO TAG1 IF U IS NIL</td>
</tr>
<tr>
<td>(PUSH 12 1)</td>
<td>SAVE U ON THE STACK</td>
</tr>
<tr>
<td>(PUSH 12 2)</td>
<td>SAVE V ON THE STACK</td>
</tr>
<tr>
<td>(HRRZ 1 0 1)</td>
<td>LOAD ACC.1 WITH CDR(U)</td>
</tr>
<tr>
<td>(CALL 2 (E INTERSECTION))</td>
<td>COMPUTE INTERSECTION(CDR(U),V)</td>
</tr>
<tr>
<td>(MOVE 2 0 12)</td>
<td>LOAD ACC.2 WITH V</td>
</tr>
<tr>
<td>(MOVEM 1 0 12)</td>
<td>SAVE INTERSECTION(CDR(U),V)</td>
</tr>
<tr>
<td>(HLRZ@ 1 -1 12)</td>
<td>LOAD ACC.1 WITH CAR(U)</td>
</tr>
<tr>
<td>(CALL 2 (E MEMBER))</td>
<td>COMPUTE MEMBER(CAR(U),V)</td>
</tr>
<tr>
<td>(EXCH 1 0 12)</td>
<td>SAVE MEMBER(CAR(U),V) AND LOAD ACC.1 WITH INTERSECTION(CDR(U),V)</td>
</tr>
<tr>
<td></td>
<td>LOAD ACC.2 WITH CAR(U)</td>
</tr>
<tr>
<td>(HLRZ@ 2 -1 12)</td>
<td>SKIP IF MEMBER(CAR(U),V) IS NOT TRUE</td>
</tr>
<tr>
<td>(SKIPE 0 0 12)</td>
<td>COMPUTE CONS(CAR(U)), INTERSECTION(CDR(U),V)</td>
</tr>
<tr>
<td>(CALL 2 (E XCONS))</td>
<td>UNDO THE FIRST TWO PUSH OPERATIONS</td>
</tr>
<tr>
<td>(SUB 12 (C 0 0 2 2))</td>
<td>RETURN</td>
</tr>
</tbody>
</table>

Tags: 0
Enter: \(R_1 = U\), \(R_2 = V\)

**EQ(U, NIL)?**

- **NO**
  - \(R_1 \times \text{CONS}(R_1, R_2)\)
  - \(R_1 \leftarrow \text{CDR}(R_1)\)
  - \(R_1 \leftarrow \text{INTERSECTION}(R_1, R_2)\)
  - \(R_2 \leftarrow \text{STACK}(0)\)
  - \(\text{STACK}(0) \leftarrow R_1\)
  - \(R_1 \leftarrow \text{CAR}(\text{STACK}(-1))\)
  - \(R_1 \leftarrow \text{MEMBER}(R_1, R_2)\)
  - \(R_1 \leftarrow \text{STACK}(0)\)
  - \(R_2 \leftarrow \text{CAR}(\text{STACK}(-1))\)

- **YES**
  - \(\text{STACK} \leftarrow R_1\)
  - \(\text{STACK} \leftarrow R_2\)

**MEMBER(\text{CAR}(U), V)?**

- **NO**
  - undo the first two stack operations
  - \(\text{RETURN}(R_1)\)

- **YES**
  - \(R_1 \leftarrow \text{XCONS}(R_1, R_2)\)

**END**
Another Example

- REVERSE function that reverses a list L
- Sample input/output: REVERSE(‘(A B C)) = ’(C B A )
- Conventional version is recursive and slow due to use of APPEND
- Use iterative (tail recursive) version REVERS1 with two arguments and vary slightly so that the result is accumulated in the first argument which enables some interesting optimizations

Initially invoked with REVERS1(NIL,L)

procedure REVERS1(RL,L)
    1   if NULL(L) then RL
    2   else REVERS1(CONS(CAR(L),RL),CDR(L))
    3   endif

- A number of possible encodings
  1. Generated by compiler
  2. Generated by hand optimization
     - Uses loop shortcutting
     - Exploits semantics of instructions that accomplish several tasks simultaneously (e.g., SKIPN)
Conventional LAP Encoding

ENTER: R1=RL  R2=L

STACK <=R1
STACK <=R2

EQ(L, NIL)?

YES

NO

R2 ← STACK(-1)
R1 ← CAR(STACK(0))
R1 ← CONS(R1,R2)
R2 ← CDR(STACK(0))
R1 ← REVERS1(R1,R2)

undo the first two stack operations
RETURN(R1)

END

PCI  (PUSH 12 I)  save RL on the stack
PC2  (PUSH 12 2)  save L on the stack
PC3  (JUMPN 2 TAG2) jump to TAG2 if L is not NIL
PC4  (JRST 0 TAGI) jump to TAG I
TAG2 (MOVE 2 -I 12) load accumulator 2 with RL
PC6  (HLRZ@ 1 0 12) load accumulator 1 with CAR(L)
        (CALL 2 (E CONS)) compute CONS(CAR(L),RL)
        (HRRZ@ 2 0 12) load accumulator 2 with CDR(L)
PC9  (CALL 2 (E REVERS1)) compute REVERSI(CONS(CAR(L),RL),CDR(L))
TAG1 (SUB 12 (C 0 0 2 2)) undo the first two push operations
PC11 (POPJ 12) return
Hand-optimized LAP Encoding

ENTER: R1 = RL             R2 = L
R3 = R2

EQ(L, NIL)?

YES
R2 ← CAR(R3)
R1 ← XCONS(R1,R2)
R3 ← CDR(R3)

NO

REVERS I (CONS (CAR (L), RL), CDR (L))

RETURN(R1)

END

COPYRIGHT 2013: HANAN SAMET

TRANSLATION VALIDATION: AUTOMATICALLY PROVING THE CORRECTNESS OF TRANSLATIONS INVOLVING OPTIMIZED CODE
Intermediate Representation (INTERSECTION)

- Use a prefix function representation

Source program

Object program

- Object program: obtained by symbolic interpretation

**Differences**

1. U and NIL may be used interchangeably
2. The symbolic intermediate representation does not indicate other differences that are present
   - INTERSECTION(CDR(U),V) is only calculated once in the object program while the source program calls for calculating it twice
   - INTERSECTION(CDR(U),V) is calculated before MEMBER(CAR(U),V) in the object program while the source program calls for its computation after MEMBER(CAR(U),V)
Example Instruction Descriptions

**HLRZ**

FEXPR HLRZ(ARGS);
LOADSTORE(ACFIELD(ARGS),
    EXTEXTDZERO(
        LEFTCONTENTS(
            EFFECTADDRESS(ARGS)));

**POPJ**

FEXR POPJ(ARGS);
BEGIN
    NEW LAB;
    LAB ← RIGHTCONTENTS(
        RIGHTCONTENTS(ACFIELD(ARGS)));
    DEALLOCATESTACKENTRY(ACFIELD(ARGS));
    SUBX(<ACFIELD(ARGS),X11>);
    UNCONDITIONALJUMP(LAB);
END
Example Instruction Descriptions

JUMPE

FEXPR JUMPE(ARGS);
BEGIN
    NEW TST;
    TST ← CHECKTEST(CONTENTS(ACFIELD(ARGS)), ZEROCNST);
    IF TST THEN RETURN(
        IF CDR (TST) THEN
            UNCONDITIONALJUMP(EFFECTADDRESS(ARGS))
        ELSE NEXTINSTRUCTION()
    );
    TRUEPREDICATE():
        CONDITIONALJUMP(ARGS, FUNCTION JUMPTRUE);
        CONDITIONALJUMP(ARGS, FUNCTION JUMPFALSE);
    END;

    FEXPR JUMPTRUE(ARGS);
    UNCONDITIONALJUMP(EFFECTADDRESS(ARGS));

    FEXPR JUMPFALSE(ARGS);
    NEXTINSTRUCTION();
Proof Process

- Must prove that no side-effect computations (e.g., an operation having the effect of a RPLACA or RPLACD in LISP) can occur between the instance of computation of INTERSECTION(CDR(U),V) and the time at which it is instantiated.
- May need to perform flow analysis.
- Some conflicts are resolved through the use of an additional intermediate representation that captures the instances of time at which the various computations were performed.
Normal Form

Normal form in terms of a tree

- Obtained through use of following axioms:
  1. $(P \rightarrow A, A) \iff_w A$
  2. $(T \rightarrow A, B) \iff A$
  3. $(\text{NIL} \rightarrow A, B) \iff B$
  4. $(P \rightarrow T, \text{NIL}) \iff P$
  5. $(P \rightarrow (P \rightarrow A, B), C) \iff (P \rightarrow A, C)$
  6. $(P \rightarrow A, (P \rightarrow B, C)) \iff (P \rightarrow A, C)$
  7. $((P \rightarrow Q, R) \rightarrow A, B) \iff (P \rightarrow (Q \rightarrow A, B), (R \rightarrow A, B))$
  8. $(P \rightarrow (Q \rightarrow A, B), (Q \rightarrow C, D)) \iff (Q \rightarrow (P \rightarrow A, C), (P \rightarrow B, D))$

- Based on McCarthy’s 1963 paper and shown by Samet in Information Processing Letters 1978 to hold for both weak and strong equivalence thereby not needing an additional pair of axioms
Distributive Law for Functions

Example:

procedure UNION(U, V)
  if NULL(U) then NIL
  else UNION(CDR(U),
    if MEMBER(CAR(U), V) then V
    else CONS(CAR(U), V))
  endif
endif

Intermediate representation reflects factoring of MEMBER test

MEMBER is encountered at a higher level in the tree than CDR(U)

Make use of an additional intermediate representation which assigns numbers to the original function representation so that as the distributive law is applied, the relative order in which the various computations are performed is not overlooked
Normal Form Algorithm

- Algorithm has two phases:
  1. Apply axioms 2, 3, and 7 along with the distributive law for functions, and also bind variables to their proper values
    - 2. $(T \rightarrow A, B) \iff A$
    - 3. $(NIL \rightarrow A, B) \iff B$
    - 7. $((P \rightarrow Q, R) \rightarrow A, B) \iff (P \rightarrow (Q \rightarrow A, B), (R \rightarrow A, B))$
  2. Apply axioms 2, 3, 5 and 6 to get rid of duplicate occurrences of predicates as well as redundant computations
    - 2. $(T \rightarrow A, B) \iff A$
    - 3. $(NIL \rightarrow A, B) \iff B$
    - 5. $(P \rightarrow (P \rightarrow A, B), C) \iff (P \rightarrow A, C)$
    - 6. $(P \rightarrow A, (P \rightarrow B, C)) \iff (P \rightarrow A, C)$
Renumbering

- Step 2 means that whenever two functions have identical computation numbers, then they must have been computed simultaneously (i.e., with the same input conditions and identical parameter bindings).
- Useful for common subexpression elimination.
- Example

```
(14 5 0)
6
(32 (28 (24 5) 6) 0)
```

- 44 is associated with two instances of UNION which yield different results as the second argument is bound to V in the first case and to '(CONS (CAR U) V)' in the second case.
- Solution is to renumber and in the process also preserve the property that each computation has a number greater than the numbers associated with its predecessors and less than those associated with its successors.
Proof

Process:
1. Transform each of the intermediate representations into the other
2. Prove that each computation appearing in one of the representations appears in the other representation and vice versa

Method:
1. Uniformly assign the computation numbers in one representation, say B, to be higher than all of the numbers in the other representation, say A, and then in increasing order, search B for matching instances of computations appearing in A
2. Reverse the above process
3. Make liberal use of axioms 1, 2, 3, 5, and 6 as well as substitution of equals for equals
4. Axiom 8 allows rearranging of condition tests if necessary
5. Make use of a sophisticated algorithm for proving equalities and inequalities of instances of formulas with function application rather than just constant symbols
Example Proof

- INTERSECTION

```
(0 10 5 0)
(16 (14 12 5) 6) 0)
(20 (18 5) 6) (26 (12 5) (24 (22 5) 6))
```

source program

```
(5 28 5 0)
(38 (36 (34 5) 6) 0)
(32 (30 5) 6) (40 (34 5) (32 (30 5) 6))
```

object program

- Must prove that \((\text{INTERSECTION} \ (\text{CDR} \ U) \ V)\) can be computed simultaneously and before the test \((\text{MEMBER} \ (\text{CAR} \ U) \ V)\)

- In other words, \((20 \ (18 \ 5) \ 6)\) and \((24 \ (22 \ 5) \ 6)\) will be shown to be matched by \((32 \ (30 \ 5) \ 6)\)

- Therefore, we prove that the act of computing \((\text{MEMBER} \ (\text{CAR} \ U) \ V)\) can be postponed to a point after computing \((\text{INTERSECTION} \ (\text{CDR} \ U) \ V)\)

- Same proof process is repeated with all computations in the object program having computation numbers less than those in the source program so that there are no computations performed in the object program that do not appear in the source program
Applications

1. Postoptimization component of a compiler
2. Interactive optimization process where a user applies transformations

3. Correctness of bootstrapping process
   - Suppose have a LISP interpreter available and want a compiler
   - Write a compiler $C$ in LISP and let the compiler translate itself yielding
   $C'$ written in assembly language
   - Proof system can be used to prove that $C$ and $C'$ are equivalent and that they generate equivalent code
   - Same process can be used if $C$ runs on machine $A$ generating code for machine $B$ and now compilers on $A$ and $B$ are equivalent

4. Bootstrapping correctness must be treated with caution as different machine architectures can cause problems with respect to different word sizes, character formats, input-output primitives, etc.

5. Found use in verifying optimizations that result in improvements in runtime behavior by reducing number of active pointers thereby increasing the amount of storage that is garbage collected
Concluding Remarks

1. Challenge was handling EQ(A,B) implies EQ(F(A),F(B))

2. Adapt to other high level languages and architectures

3. Recursion is the only control flow mechanism
   - Interpret recursion as having taken place whenever symbolic interpretation process encounters an instruction which has been encountered previously along the same path (termed loop shortcutting)

4. Could handle GO in LISP by breaking up program into modules of intervals having one entry point and several exit points
   - Branches which jump back anywhere within the interval other than the entry point are interpreted as instances of loop shortcutting
   - Branches to points other than entry nodes in other intervals are also interpreted as instances of loop shortcutting
   - Need a proof for each interval

5. Potential drawback is that intermediate representation in the form of a tree with $N$ conditions could grow as big as $2^N$ execution paths
   - But COND (if-then-else) of $N$ conditions only has $N + 1$ execution paths


Historical References


Historical References (Continued)


Proof-Carrying Code References


Translation Validation References


References Acknowledging Samet’s Contributions


