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

http://www.cs.umd.edu/~hjs/pubs/compilers/CS-TR-75-498.pdf
http://www.cs.umd.edu/~hjs/slides/translation-validation.pdf

All text in blue can be clicked to get more information

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:
  1. Specification of the assertions
  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 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)

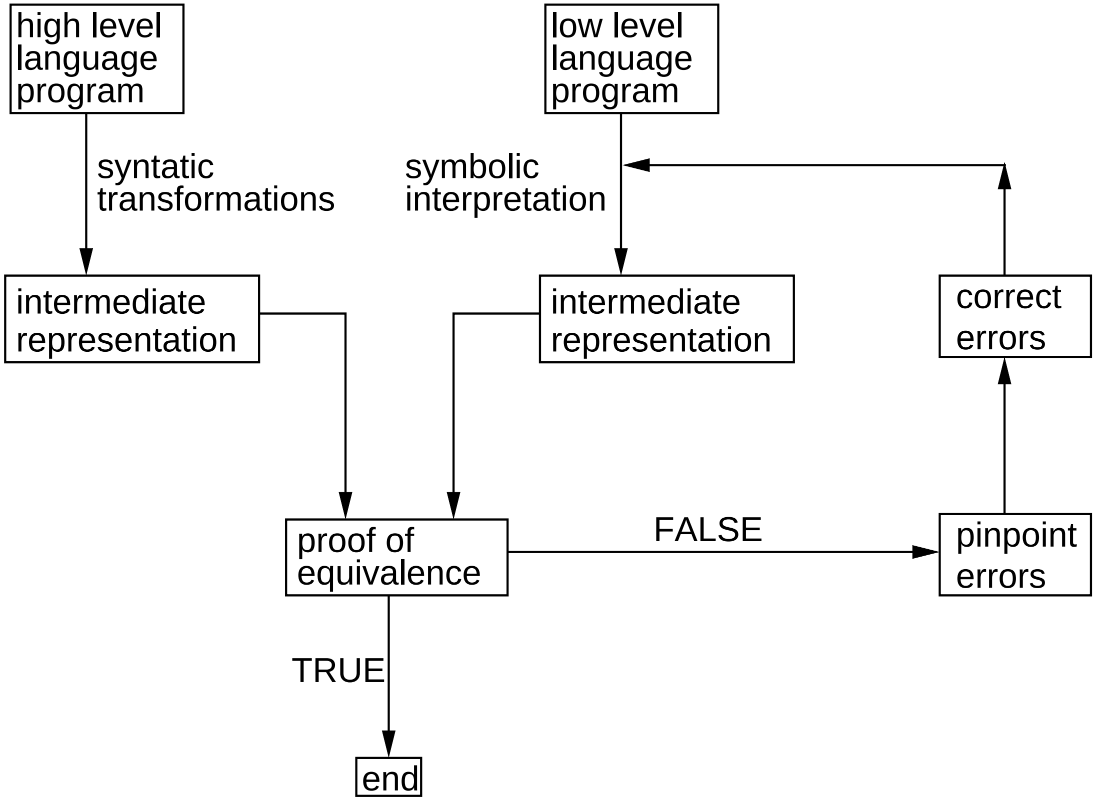
Historical Perspective I

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 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
  1. University of York in UK: Stringer-Calvert (1998) and Buttle (1998, 2001) who also mention the work of Pavey and Winsborrow (1995) for the verification of a protection system for a nuclear power station
  2. Germany: Zimmerman (2004, 2006), Leinenbach (2008), Gawkowski (2008), Blech (2009), and Blech and Gregoire (2011)
  3. France: Tristan (2008), Rideau and Leroy (2010), Jourdan, Pottier, and Leroy (2012)
  4. Belgium: Shashidhar (2008)
  5. US: Kundu Tatlock, and Lerner (2009); Tate, Stepp, Tatlock, and Lerner (2009,2011); Tatlock and Lerner (2010); Stepp, Tate, and Lerner (2011)

Compiler Testing System Architecture



- Equivalence proof applies equivalence preserving transformations in an attempt to reduce them to a common representation termed a <u>normal form</u>
- 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

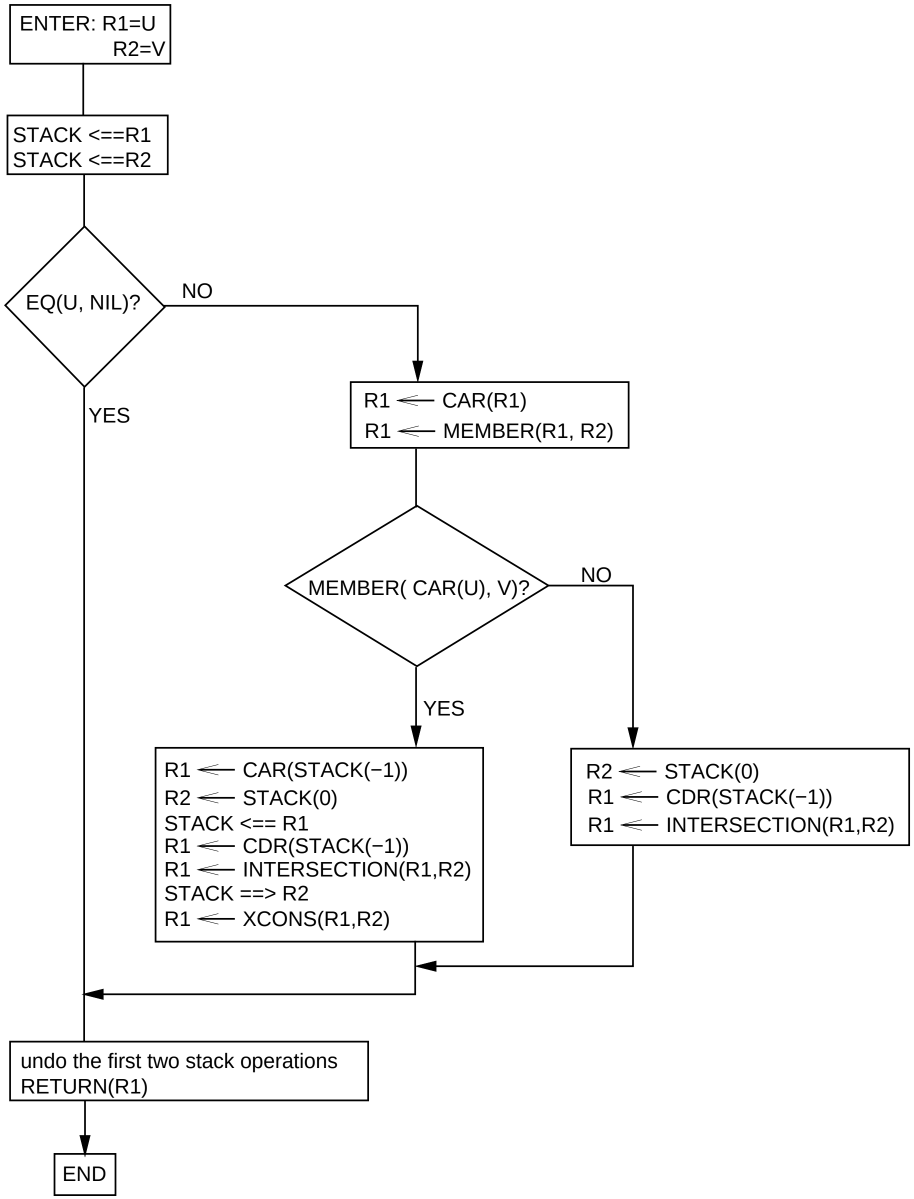
Example

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

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

- Sample input/output: INTERSECTION(’(A B C),’(D C B)) = ’(B C)

Flowchart of Conventional LAP Encoding

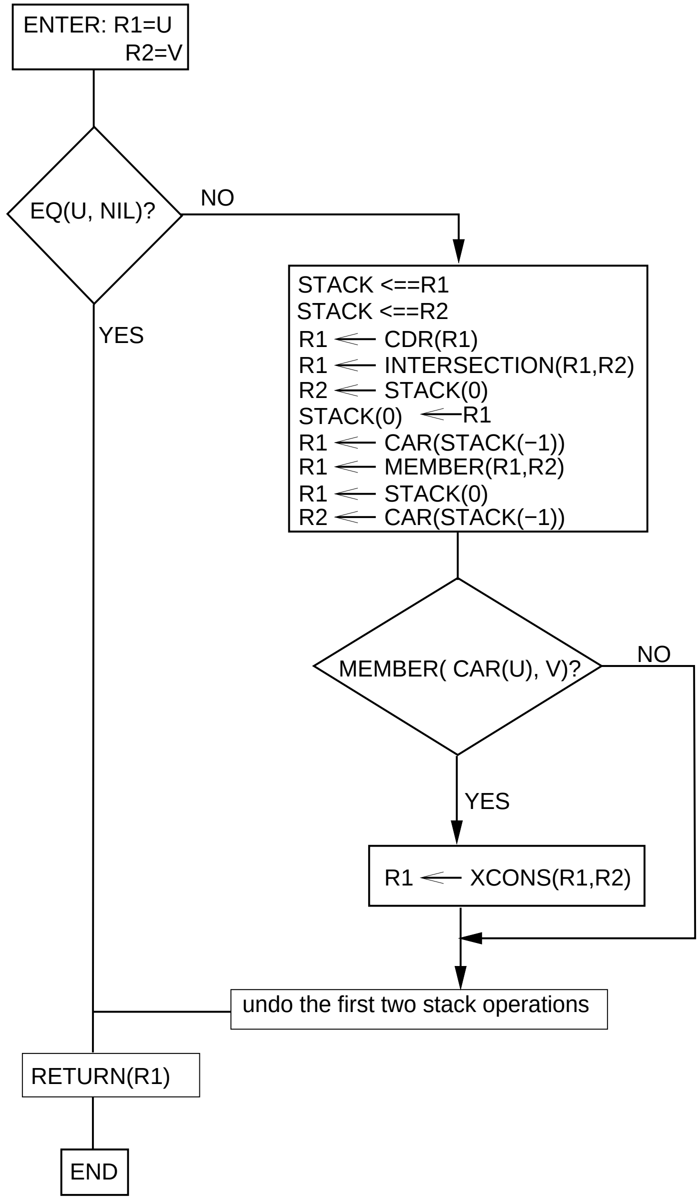


Example Optimized LAP Encoding

- Obtained by hand optimization process

INTERSECTION	(JUMPE 1 TAG 1)	JUMP TO TAG1 IF U IS NIL
	(PUSH 12 1)	SAVE U ON THE STACK
	(PUSH 12 2)	SAVE V ON THE STACK
	(HRRZ 1 0 1)	LOAD ACC.1 WITH CDR(U)
	(CALL 2 (E INTERSECTION))	COMPUTE INTERSECTION(CDR(U),V)
	(MOVE 2 0 12)	LOAD ACC.2 WITH V
	(MOVEM 1 0 12)	SAVE INTERSECTION(CDR(U),V)
	(HLRZ@ 1 −1 12)	LOAD ACC.1 WITH CAR(U)
	(CALL 2(E MEMBER))	COMPUTE MEMBER(CAR(U),V)
	(EXCH 1 0 12)	SAVE MEMBER(CAR(U),V) AND LOAD ACC.1 WITH INTERSECTION(CDR(U),V)
	(HLRZ@ 2 −1 12)	LOAD ACC.2 WITH CAR(U)
	(SKIPE 0 0 12)	SKIP IF MEMBER(CAR(U),V) IS NOT TRUE
	(CALL 2(E XCONS))	COMPUTE CONS(CAR(U)), INTERSECTION(CDR(U),V)
	(SUB 12(C 0 0 2 2))	UNDO THE FIRST TWO PUSH OPERATIONS
TAG1	(POPJ 12)	RETURN

Flowchart of Optimized LAP Encoding



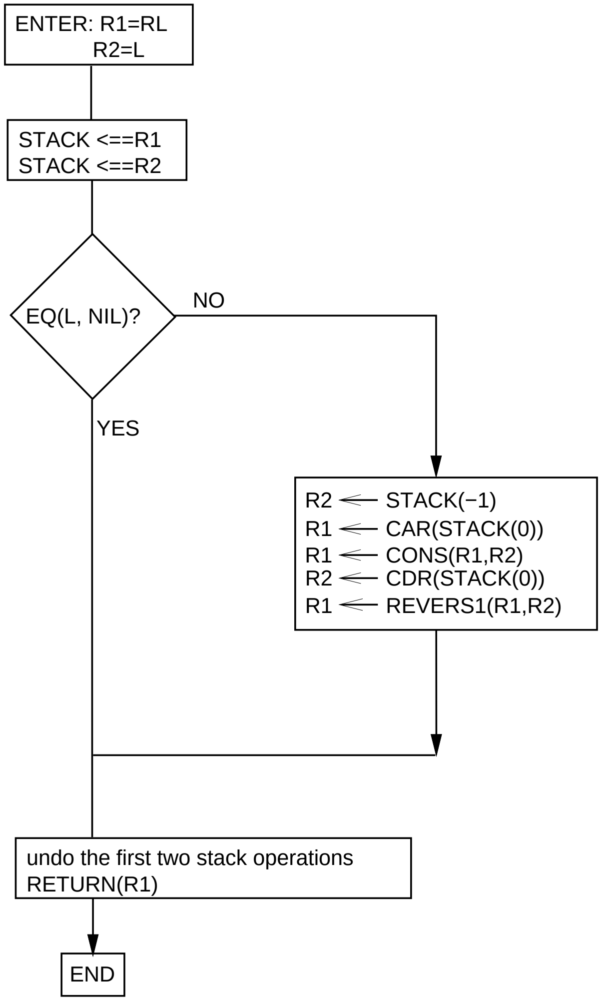
Another Example

- 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)
    if NULL(L) then RL
    else REVERS1(CONS(CAR(L),RL),CDR(L))
    endif
```

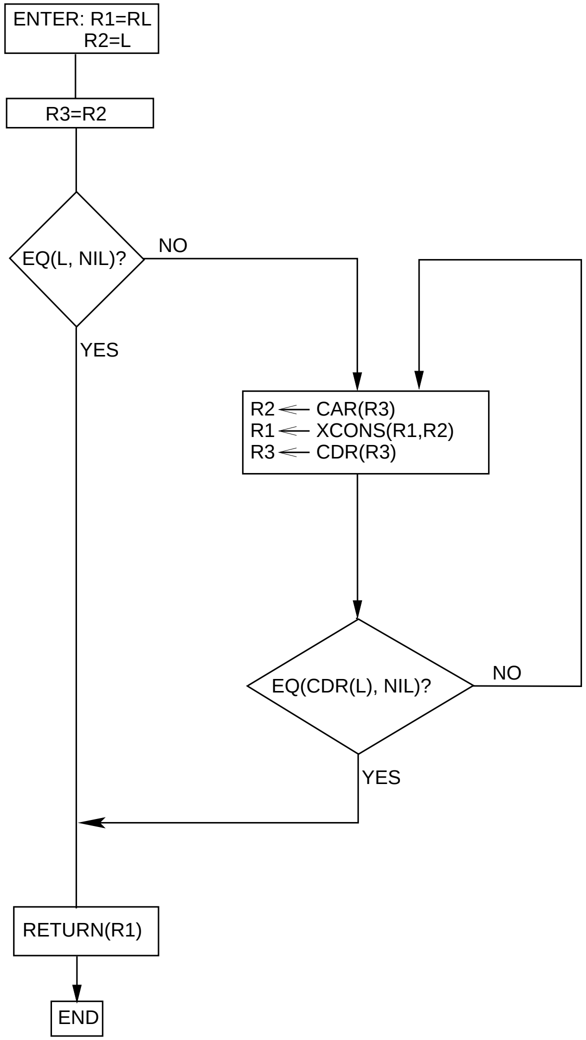
- A number of possible encodings
  1. Generated by compiler
  2. Generated by hand optimization
     - Uses loop shortcutting
     - Exploits semantics of instructions that accomplish several tasks simultaneously (e.g., SKIPN)

Conventional LAP Encoding



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@ I 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

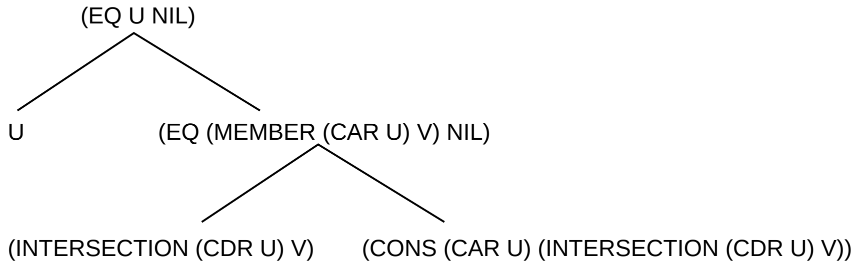
Hand-optimized LAP Encoding



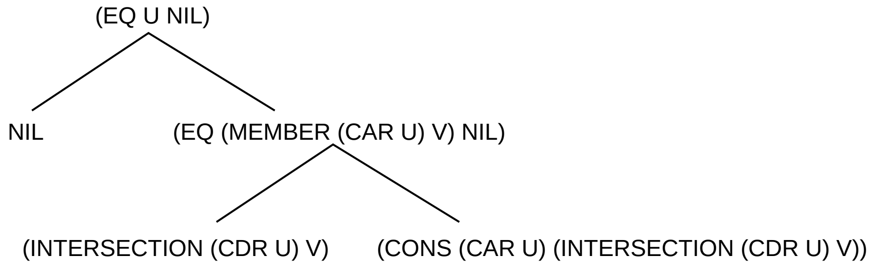
	(SKIPN 3 2)	load accumulator 3 with L and skip if not NIL
	(POPJ 12)	return NIL
REV	(HLRZ 2 0 3)	load accumulator 2 with CAR(L)
	(CALL 2 (E XCONS))	compute CONS (CAR(L),RL)
	(HRRZ 3 0 3}	load accumulator 3 with CDR(L)
	(JUMPN 3 REV)	if CDR(L) is not NIL then compute REVERS I (CONS (CAR (L), RL), CDR (L))
	(POPJ 12)	return

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),
              EXTEXDZERO(
                      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

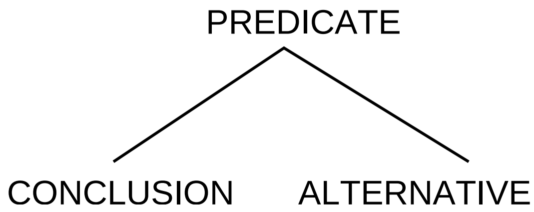
```
FEXPR JUMPE(ARGS);
BEGIN
    NEW TST;
    TST ← CHECKTEST(CONTENTS(ACFIELD(ARGS)),ZEROCNST);
    IF TST THEN RETURN(
        IF CDR (TST) THEN
                UNCONDITIONALJUMP(EFFECTADDRESS(ARGS))
        ELSE NEXTINSTRUCTION());
    TRUEPREDICATE():
    CONDITIONALJUMP(ARGS,FUNCTION JUMPTRUE);
    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. $(P \rightarrow A, A) \Longleftrightarrow_w A$
  2. $(T \rightarrow A, B) \Longleftrightarrow A$
  3. $(NIL \rightarrow A, B) \Longleftrightarrow B$
  4. $(P \rightarrow T, NIL) \Longleftrightarrow P$
  5. $(P \rightarrow (P \rightarrow A, B), C) \Longleftrightarrow (P \rightarrow A, C)$
  6. $(P \rightarrow A, (P \rightarrow B, C)) \Longleftrightarrow (P \rightarrow A, C)$
  7. $((P \rightarrow Q, R) \rightarrow A, B) \Longleftrightarrow (P \rightarrow (Q \rightarrow A, B), (R \rightarrow A, B))$
  8. $(P \rightarrow (Q \rightarrow A, B), (Q \rightarrow C, D)) \Longleftrightarrow (Q \rightarrow (P \rightarrow A, C), (P \rightarrow B, D))$
- 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

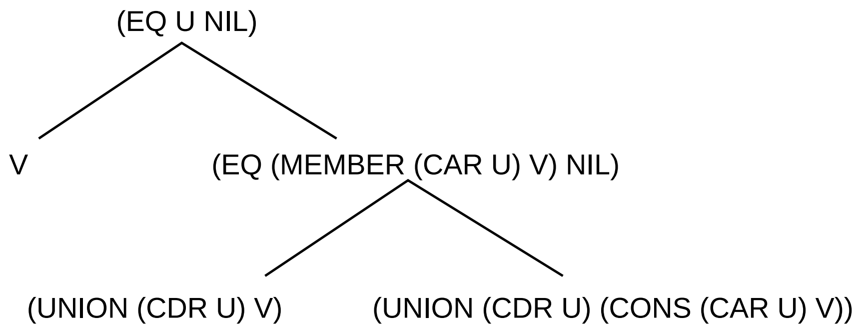
Distributive Law for Functions

- Example:

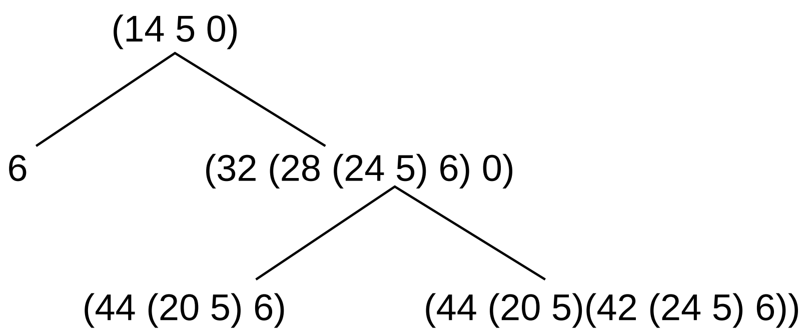
  procedure UNION(U,V)
  if NULL(U) then NIL
  else UNION(CDR(U),

  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

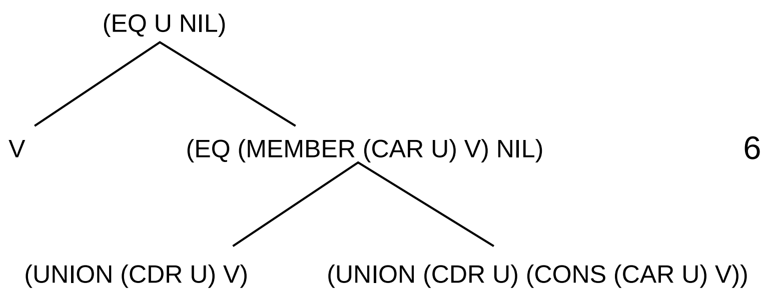


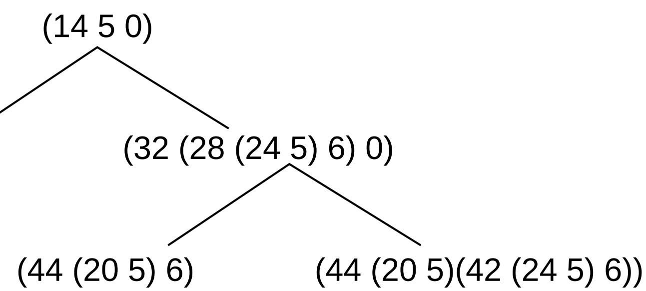
Normal Form Algorithm

- 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. $(T \rightarrow A,B) \Longleftrightarrow A$
     - 3. $(NIL \rightarrow A,B) \Longleftrightarrow B$
     - 7. $((P \rightarrow Q,R) \rightarrow A,B) \Longleftrightarrow (P \rightarrow (Q \rightarrow A,B),(R \rightarrow A,B))$
  2. Apply axioms 2, 3, 5 and 6 to get rid of duplicate occurrences of predicates as well as redundant computations
     - 2. $(T \rightarrow A,B) \Longleftrightarrow A$
     - 3. $(NIL \rightarrow A,B) \Longleftrightarrow B$
     - 5. $(P \rightarrow (P \rightarrow A,B),C) \Longleftrightarrow (P \rightarrow A,C)$
     - 6. $(P \rightarrow A,(P \rightarrow B,C)) \Longleftrightarrow (P \rightarrow A,C)$

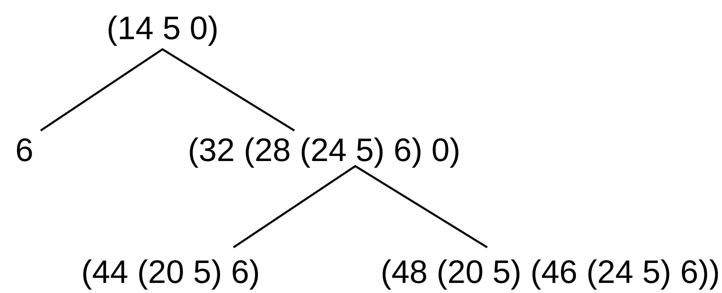
Renumbering

- 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





- 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

source program | object program

- Must prove that (INTERSECTION (CDR U) V) can be computed simultaneously and before the test (MEMBER (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 (MEMBER (CAR U) V) can be postponed to a point after computing (INTERSECTION (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))
   - Uniform word problem (see Samet 1974, Samet 1977 TR, Downey et al. 1978, Samet 1980 IEEETC)
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

Samet References (Click on Blue Titles for PDFs)

1. H. Samet. Equivalence and inequivalence of instances of formulas (unpublished). Computer Science Department, Stanford University, Stanford, CA, April 1974.
2. H. Samet. Automatically proving the correctness of translations involving optimized code. PhD thesis, Computer Science Department, Stanford University, Stanford, CA, May 1975. Also Technical Report - CS-TR-75-498 (Warning pdf size-58MB).
3. H.Samet. Increasing the reliability of code generation. In Proceedings of the Fourth International Conference on the Implementation Design of Algorithmic Languages, pages 193–203, New York, June 1976.
4. H.Samet. Compiler testing via symbolic interpretation. In Proceedings of the ACM 29th Annual Conference, pages 492–497, Houston, TX, October 1976.
5. H.Samet. Towards code optimization in LISP. In Proceedings of the 5th International Conference on the Implementation and Design of Algorithmic Languages, pages 362–374, Rennes, France, May 1977.

Samet References (Continued)

6. H.Samet. A normal form for compiler testing. In Proceedings of the SIGART SIGPLAN Symposium on Artificial Intelligence and Programming Languages, pages 155–162, Rochester, NY, Aug 1977. Also in SIGPLAN NOTICES, August 1977 and in SIGART NEWSLETTER, August 1977.
7. H. Samet. Equivalence and inequivalence of instances of formulas. Computer Science Technical Report TR–553, University of Maryland, College Park, MD, August 1977.
8. H. Samet. Toward automatic debugging of compilers. In Proceedings of the 5th International Joint Conference on Artificial Intelligence, page 379, Cambridge, MA, August 1977.
9. H. Samet. A machine description facility for compiler testing. IEEE Transactions on Software Engineering, 3(5):343–351, September 1977. Also in Computing Reviews, 19(3):113–114, entry 32738 March 1978.
10. H. Samet. A new approach to evaluating code generation in a student environment. In Information Processing 77, B. Gilchrist, ed., pages 661–665. North Holland, Toronto, Canada, 1977.

Samet References (Continued)

11. P. J. Downey, H. Samet and R. Sethi. Off-line and on-line algorithms for deducing equalities. In Proceedings of the 5th ACM Symposium on Principles of Programming Languages (POPL'78), A. V. Aho, S. N. Zilles, and T. G. Szymanski, eds., pages 158–170, Tucson, AZ, January 1978. Also in Computing Reviews, 20(4):157, entry 34427, April 1979.
12. H.Samet. A canonical form algorithm for proving equivalences of conditional forms. Information Processing Letters, 7(2):103–106, February 1978.
13. H.Samet. Proving correctness of heuristically optimized code. Communications of the ACM, 21(7):570–582, July 1978.
14. H.Samet. Efficient on-line proofs of equalities and inequalitiesof formulas. IEEE Transactions on Computers, 29(1):28–32, January 1980.
15. H.Samet and L.Marcus. Purging in an equality data base. Information Processing Letters, 10(2):89–95, March 1980.
16. H.Samet. Experience with software conversion. Software - Practice and Experience, 11(10):1053–1069, 1981.
17. H. Samet. Code optimization considerations in list processing systems. IEEE Transactions on Software Engineering, 8(2):107–112, March 1982.

Historical References

1. J. McCarthy. A basis for a mathemtical theory of computation. In Computer Programming and Formal Systems, P. Braffort and D. Hershberg, eds., vol. 35, pages 33–70. North Holland, Amsterdam, The Netherlands, 1963.
2. J. McCarthy and J. Painter. Correctness of a compiler for arithmetic expressions. In Mathematical Aspects of Computer Science, J. T. Schwartz, ed., vol. 19 of Proceedings of Symposia in Applied Mathematics, pages 33–41, American Mathematical Society, 1967.
3. R. Milner. An algebraic definition of simulation between programs. In Proceedings of the 2nd International Joint Conference on Artificial Intelligence, D. C. Cooper, ed., pages 481–489, London, United Kingdom, September 1971.
4. G. Kreisel. II. Checking of computer programs: An example of non-numerical computation. In Five Notes on the Application of Proof Theory to Computer Science, pages 15–21. Technical Report No. 182, Institute for Mathematical Studies in the Social Sciences, Stanford University, Stanford, CA, December 1971.

Historical References (Continued)

5. R. Milner and R. Weyhrauch. Proving compiler correctness in a mechanized logic. In Proceedings of 7th Annual Machine Intelligence Workshop, B. Meltzer and D. Michie, eds., vol. 7 of Machine Intelligence, pages 51–72, Edinburgh University Press, 1972.
6. D.B. Krafft and A.J. Demers. Determining logical dependency in a decision procedure for equality. Computer Science Technical Report TR–81–458, Cornell University, Ithaca, NY, April 1981.
7. M. Blum and S. Kannan. Designing programs that check their work. In Proceedings of the 21st Annual ACM Symposium on the Theory of Computing, pages 86–97, Seattle, May 1989.
8. M. Blum and S. Kannan. Designing programs that check their work. Journal of the ACM, 42(1):269–291, January 1995.
9. D.J. Pavey and L.A. Winsborrow. Demonstrating equivalence of source code and prom contents. The Computer Journal, 36(7):654–667, January 1993.

Proof-Carrying Code References

1. G. C. Necula and P. Lee. Safe kernel extensions without run-time checking. In Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation (OSDI), K. Petersen and W. Zwaenepoel, eds., pages 229–243, Seattle, WA, October 1996.
2. G. C. Necula and P. Lee. Proof-carrying code. Technical Report CMU–CS–96–165, School of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, November 1996.
3. G. C. Necula. Proof-carrying code. In Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97), P. Lee, F. Henglein, and N. D. Jones, eds., pages 106–119, Paris, France, January 1997.
4. G. C. Necula. Translation validation for an optimizing compiler. In Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), M. S. Lam, ed., pages 83–94, Vancouver, Canada, June 2000.

Translation Validation References

1. A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS98), Steffen B, ed., vol. 1384 of Springer-Verlag Lecture Notes in Computer Science, pages 151–166, Lisbon, Portugal, March 1998.
2. L. D. Zuck, A. Pnueli, Y. Fang, and B. Goldberg. VOC: A methodology for the translation validation of optimizing compilers. Journal of Universal Computer Science, 9(3):223–247, 2003.

References Acknowledging Samet's Contributions

1. D. W. J. Stringer-Calvert. Mechanical verification of compiler correctness PhD thesis, Department of Computer Science, University of York, York, UK, March 1998.
2. D. L. Buttle. Verification of compiled code PhD thesis, Department of Computer Science, University of York, York, UK, January 2001.
3. W. Zimmerman. On the correctness of transformations in compiler back-ends. In Proceedings of the First International Symposium on the Leveraging of Formal Methods, vol. 4313 of Springer-Verlag Lecture Notes in Computer Science, pages 74–95, Paphos, Cyprus, October 2004.
4. Shashidhar. Efficient automatic verification of loop and data-flow transformations by functional equivalence checking. PhD thesis, Department of Computer Science, Katholieke Universiteit Leuven, Leuven, Belgium, May 2008.
5. D. C. Leinenbach. Compiler verification in the context of pervasive system verification. PhD thesis, Naturwissenschaftlich-Technische Fakultat I, Saarlandes University, Saarbrucken, Germany, 2008.

References Acknowledging Samet (Continued)

6. M. J. Gawkowski. Formal framework for proof generating optimizers. PhD thesis, Fachbereich Informatik, Kaiserslautern University, Kaiserslautern, Germany, November 2008.
7. J.-O. Blech. Certifying system translations using higher order theorem provers. PhD thesis, Fachbereich Informatik, Kaiserslautern University, Kaiserslautern, Germany, 2009.
8. J.-O. Blech and B. Grégoire. Certifying compilers using higher-order theorem provers as certificate checkers. Formal Methods in System Design, 38(1):33–61, February 2011.
9. J.-B. Tristan. Formal verification of translation validators. PhD thesis, Graduate School of Mathematical Science of Paris Diderot (Paris 7), Paris, France, November 2009.
10. S. Rideau and X. Leroy. Validating register allocation and spilling. In Proceedings of the 19th International Conference on Compiler Construction (CC2010), R. Gupta, ed., vol. 6011 of Springer-Verlag Lecture Notes in Computer Science, pages 86–97, Paphos, Cyprus, March 2010.

References Acknowledging Samet (Continued)

11. J.-H. Jourdan, F. Pottier, and X. Leroy. Validating LR(1) parsers. In Programming Languages and Systems–Proceedings of the 21st European Symposium on Programming (ESOP12), H. Seidl, ed., vol. 7211 of Springer-Verlag Lecture Notes in Computer Science, pages 397–416, Tallinn, Estonia, March 2012.
12. R. Tate, M. Stepp, Z. Tatlock, and S. Lerner. Equality saturation: a new approach to optimization. In Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'09), Z. Shao and B. C. Pierce, eds., pages 264–276, Savannah, GA, January 2009.
13. S. Kundu, Z. Tatlock, and S. Lerner. Proving optimizations correct using parameterized program equivalence. In Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), M. Hind and A. Diwan, eds., pages 327–337, Dublin, Ireland, June 2009.

References Acknowledging Samet (Continued)

14. M. Stepp, R. Tate, and S. Lerner. Equality-based translation validator for LLVM. In Computer Architectures for Spatially Distributed Data, G. Gopalakrishnan and S. Qadeer, eds., vol. 6806 of Springer-Verlag Lecture Notes in Computer Science, pages 737–742, Snowbird, UT, July 2011.
15. R. Tate, M. Stepp, Z. Tatlock, and S. Lerner. Equality saturation: a new approach to optimization. Logical Methods in Computer Science, 7(1), 2011.
16. Z. Tatlock and S. Lerner. Bringing extensibility to verified compilers. In Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), B. G. Zorn and A. Aiken, eds., pages 111–121, Toronto, Canada, June 2010.

