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/dagstuhl05.pdf`

The assistance of Jagan Sankaranarayanan in making the slides is appreciated.

Compiler Testing (also now known as Translation Validation)

- 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
  1. Critical semantic properties of high level language must be identified
  2. Identify their interrelationship to instruction set of computer executing the resulting translation

Relation to Other Work

- Interested in proving that programs are correctly translated
- Different from proving that programs are correct
- 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 the purpose of the 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

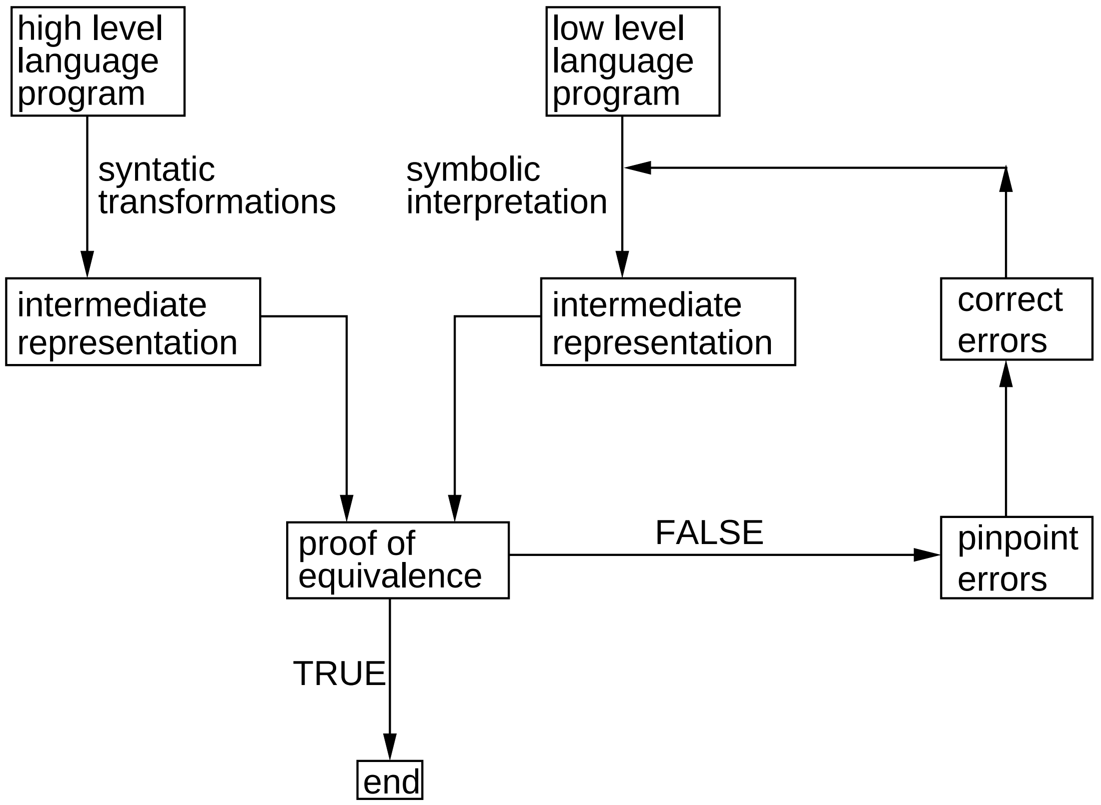
Our Approach

- 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
- In other words, they have identical execution sequences
- Therefore, they must test the same conditions except for certain valid rearrangements of computations
- We prove correctness of the translation
- Historical roots:
  - Originated by Samet in Ph.D. thesis in 1975
  - Proof Carrying Code of Necula and Lee in 1996 is closely related
  - Rediscovered by Pnueli, Siegel, and Singerman in 1998 and termed it Translation Validation and followed by Barrett, Fang, Hu, Goldberg, and Zuck
  - Acknowledgment of relationship to Samet’s work includes Blech, Gawkowski, Kundu, Lerner, Leroi, Rideau, Stepp, Tate, Tatloc, Tristan, and Zimmerman

Alternative Approaches

- One method is to prove that there does not exist a program which is incorrectly translated by the compiler
  1. Instead, we prove that for each program input to the translation process, the translated version is equivalent to the original version
  2. A proof must be generated for each input to the translation process
  3. Advantage is that as long as compiler performs its job for each program input to it, its correctness is of a secondary nature
  4. Proof system can run as a postprocessing step to compilation
  5. We have bootstrapped ourselves so that we can attribute an “effective correctness to the compiler”
  6. 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
  7. 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)

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

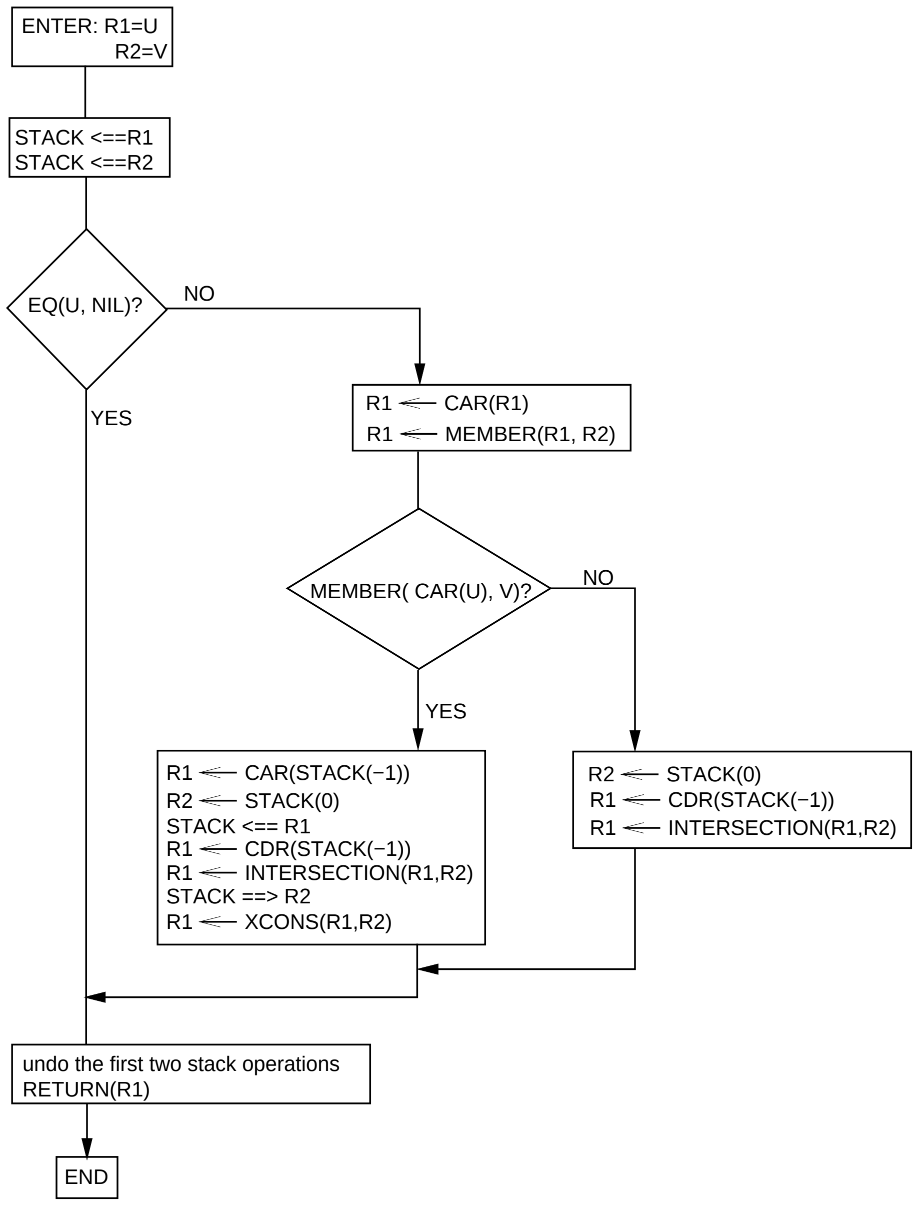
Example

- 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)
    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

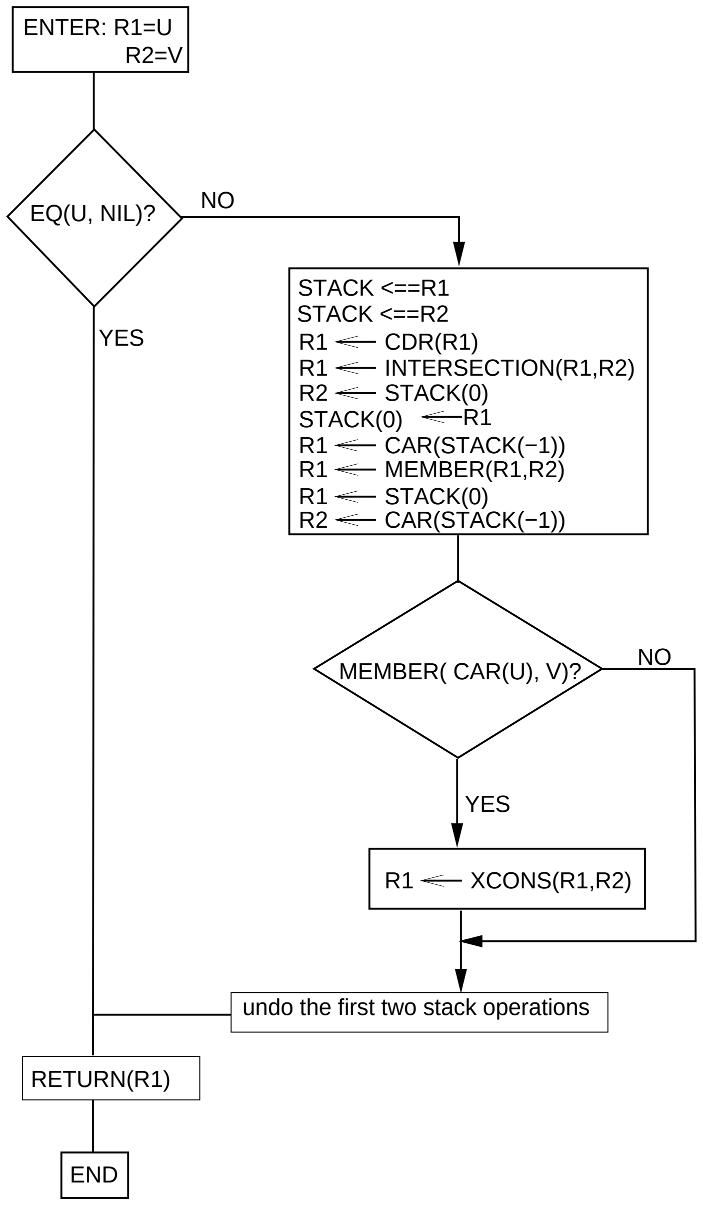


Example Optimized LAP Encoding

- 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



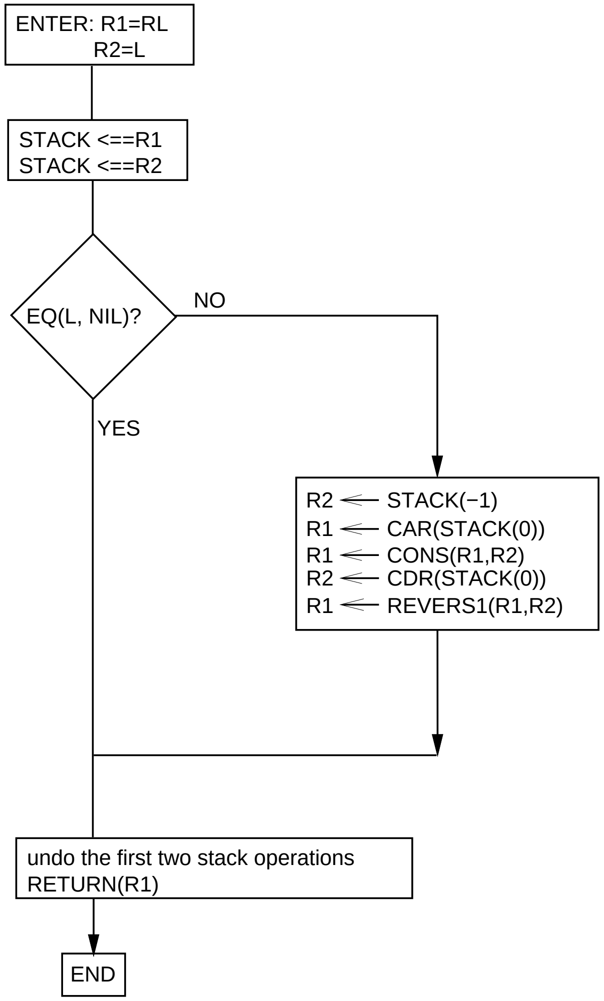
Another Example

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

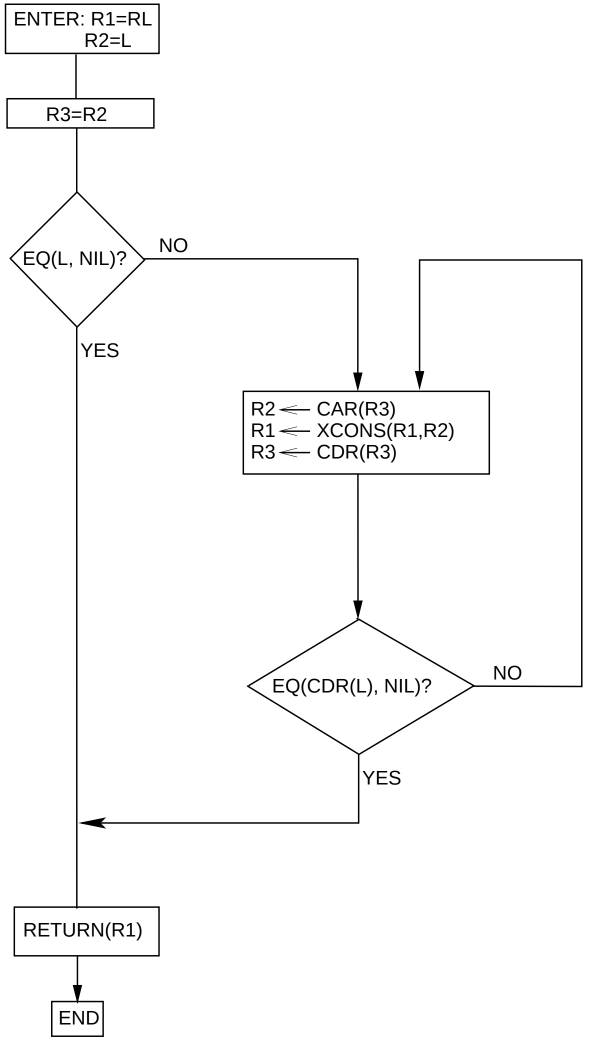
- A number of possible encodings
  1. Generated by compiler
  2. Generated by hand optimization
     - Uses loop shortcutting
     - Exploits semantics of instructions that accomplish several tasks simultaneously (e.g., SKIPN)

Conventional LAP Encoding



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

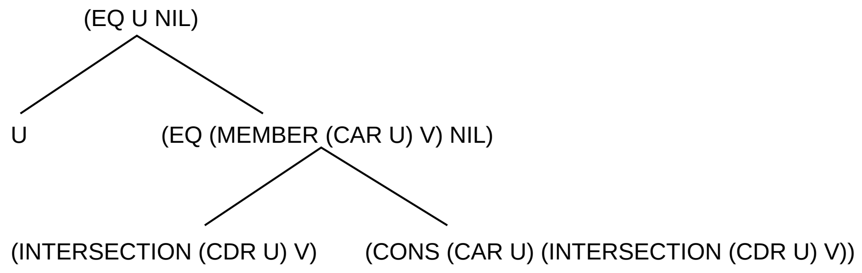
Hand-optimized LAP Encoding



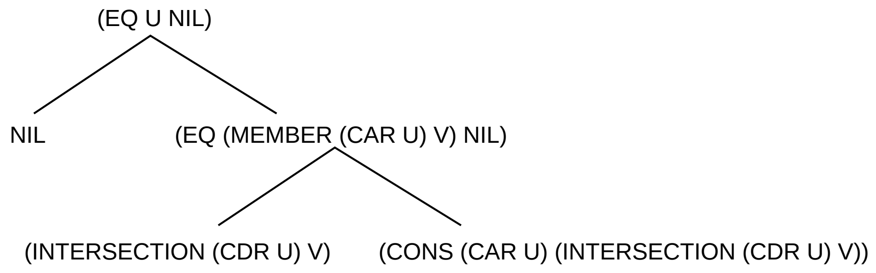
```
REV  (SKIPN 3 2)               load accumulator 3 with L and skip if not NIL
     (POPJ 12)                 return NIL
     (HLRZ 2 0 3)              load accumulator 2 with CAR(L)
     (CALL 2 (E XCONS))        compute CONS (CAR(L),RL)
     (HRRZ 3 0 3}              load accumulator 3 with CDR(L)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


(EQ U NIL)
U
(EQ (MEMBER (CAR U) V) NIL)
(INTERSECTION (CDR U) V)
(CONS (CAR U) (INTERSECTION (CDR U) V))

Source program


(EQ U NIL)
NIL
(EQ (MEMBER (CAR U) V) NIL)
(INTERSECTION (CDR U) V)
(CONS (CAR U) (INTERSECTION (CDR U) V))

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

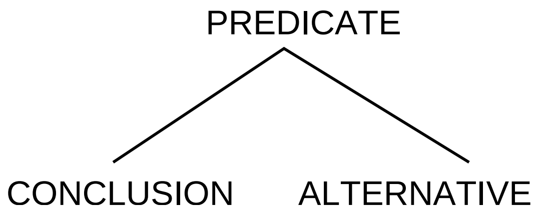
```
FEXPR JUMPE(ARGS);
BEGIN
    NEW TST;
    TST ← CHECKTEST(CONTENTS(ACFIELD(ARGS)),ZEROCNST);
     IF TST THEN RETURN(
         IF CDR (TST) THEN
                     UNCONDITIONALJUMP(EFFECTADDRESS(ARGS))
         ELSE NEXTINSTRUCTION());
     TRUEPREDICATE():
     CONDITIONALJUMP(ARGS,FUNCTION JUMPTRUE);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

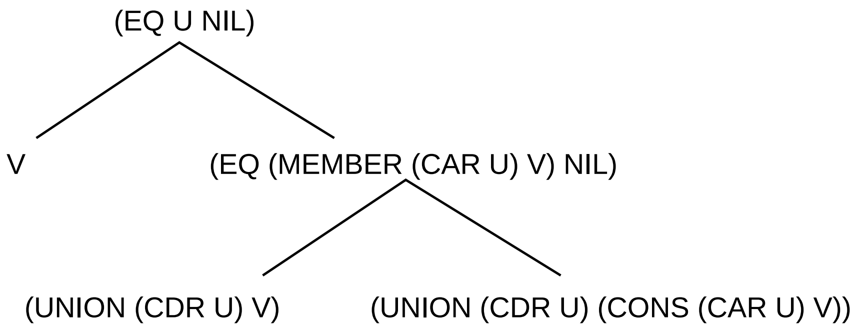


- Obtained through use of following axioms:
  1. $(\mathrm{P}\rightarrow\mathrm{A},\mathrm{A})\Longleftrightarrow_w\mathrm{A}$
  2. $(\mathrm{T}\rightarrow\mathrm{A},\mathrm{B})\Longleftrightarrow\mathrm{A}$
  3. $(\mathrm{NIL}\rightarrow\mathrm{A},\mathrm{B})\Longleftrightarrow\mathrm{B}$
  4. $(\mathrm{P}\rightarrow\mathrm{T},\mathrm{NIL})\Longleftrightarrow\mathrm{P}$
  5. $(\mathrm{P}\rightarrow(\mathrm{P}\rightarrow\mathrm{A},\mathrm{B}),\mathrm{C})\Longleftrightarrow(\mathrm{P}\rightarrow\mathrm{A},\mathrm{C})$
  6. $(\mathrm{P}\rightarrow\mathrm{A},(\mathrm{P}\rightarrow\mathrm{B},\mathrm{C}))\Longleftrightarrow(\mathrm{P}\rightarrow\mathrm{A},\mathrm{C})$
  7. $((\mathrm{P}\rightarrow\mathrm{Q},\mathrm{R})\rightarrow\mathrm{A},\mathrm{B})\Longleftrightarrow(\mathrm{P}\rightarrow(\mathrm{Q}\rightarrow\mathrm{A},\mathrm{B}),(\mathrm{R}\rightarrow\mathrm{A},\mathrm{B}))$
  8. $(\mathrm{P}\rightarrow(\mathrm{Q}\rightarrow\mathrm{A},\mathrm{B}),(\mathrm{Q}\rightarrow\mathrm{C},\mathrm{D}))\Longleftrightarrow(\mathrm{Q}\rightarrow(\mathrm{P}\rightarrow\mathrm{A},\mathrm{C}),(\mathrm{P}\rightarrow\mathrm{B},\mathrm{D}))$
- Based on McCarthy63 and shown by SametInfoPL78 to hold for both weak and strong equivalence thereby not needing an additional pair of axioms

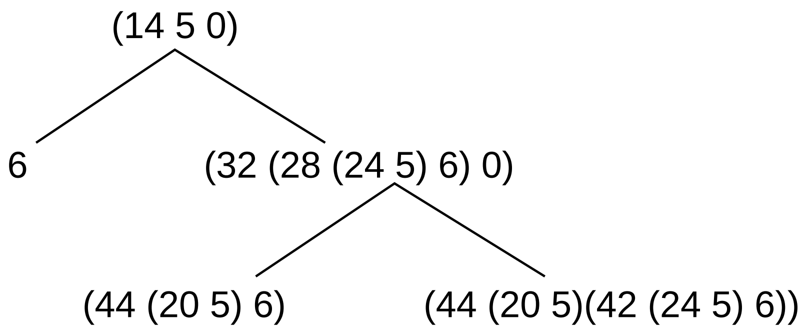
Distributive Law for Functions

- 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

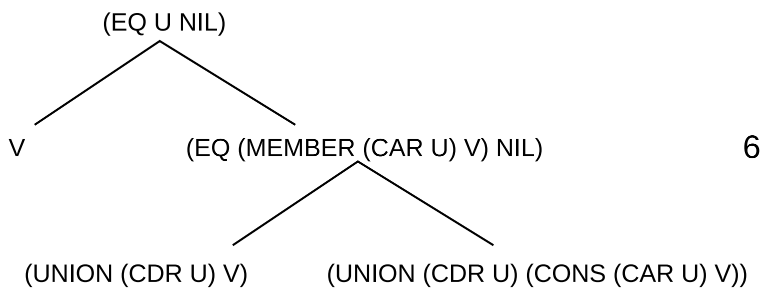


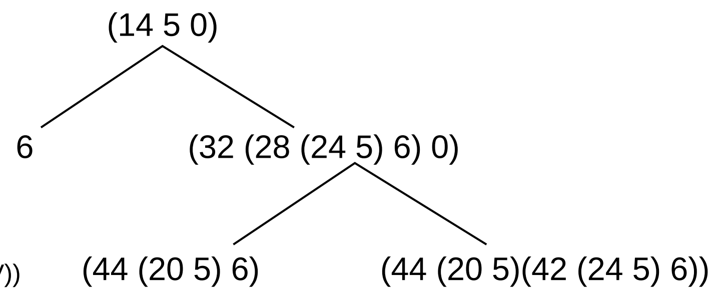
Normal Form Algorithm

- 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. $(P \rightarrow A,(P \rightarrow B,C)) \Longleftrightarrow (P \rightarrow A,C)$

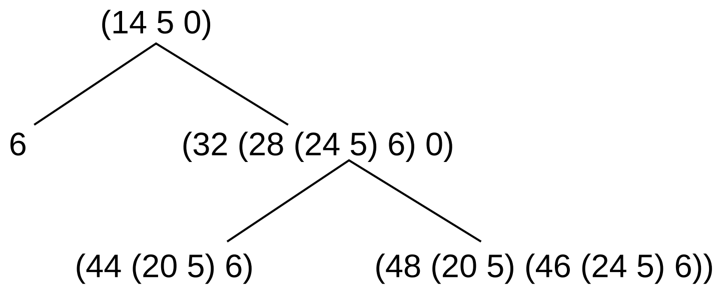
Renumbering

- 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 second 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

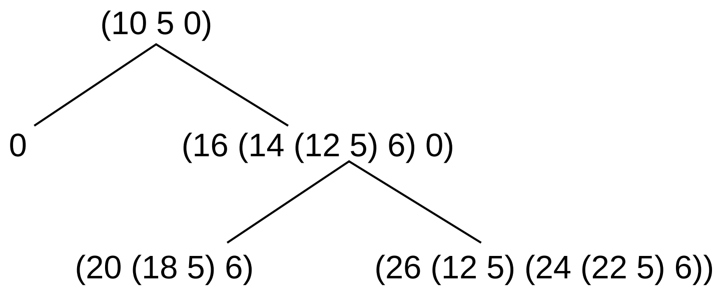


Proof

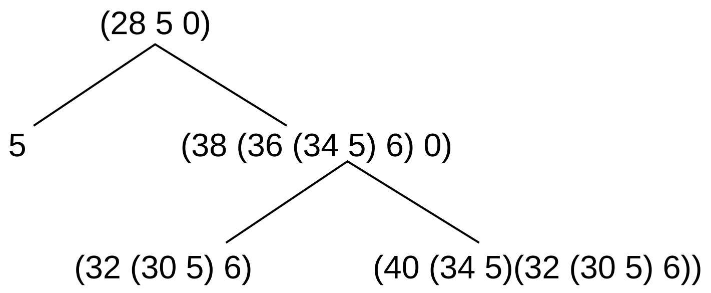
- Process:
  1. Transform each of the intermediate representations into the other
  2. Prove that each computation appearing in one of the representations appears in the other representation and vice versa
- Method:
  1. Uniformly assign the computation numbers in one representation, say B, to be higher 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 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
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 <u>loop shortcutting</u>)
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

References

1. H. Samet, Automatically Proving the Correctness of Translations Involving Optimized Code, Ph.D. thesis, Stanford University, CA, May 1975 (Also Technical Report - CS-TR-75-498, Department of Computer Science, Stanford University, CA) (Warning pdf size-58MB).
2. H.Samet, Increasing the Reliability of Code Generation, Proceedings of the Fourth International Conference on the Implementation Design of Algorithmic Languages, New York, June 1976, pages 193-203.
3. H.Samet, Compiler Testing via Symbolic Interpretation, Proceedings of the ACM 29th Annual Conference, Houston, TX, October 1976, pages 492-497.
4. H.Samet, Towards Code Optimization in LISP, Proceedings of the 5th International Conference on the Implementation and Design of Algorithmic Languages, Rennes, France, May 1977, pages 362-374.
5. H.Samet, A Normal Form for Compiler Testing, Proceedings of the SIGART SIGPLAN Symposium on Artificial Intelligence and Programming Languages, Rochester, NY, August 1977, pages 155-162, (also in SIGPLAN NOTICES, August 1977 and in SIGART NEWSLETTER, August 1977).

References (Continued)

6. H.Samet, Toward Automatic Debugging of Compilers, Proceedings of the 5th International Joint Conference on Artificial Intelligence, Cambridge, MA, August 1977, page 379.
7. H.Samet, A Machine Description Facility for Compiler Testing, IEEE Transactions on Software Engineering 3, 5(September 1977), pages 343-351 (also in Computing Reviews 19, 3(March 1978), pages 113-114, entry 32738).
8. H.Samet, A New Approach to Evaluating Code Generation in a Student Environment, Information Processing 77, (B. Gilchrist, Ed.), North Holland Publishing Company, 1977, pages 661-665.
9. P.J. Downey, H.Samet and R. Sethi, Off-line and On-line Algorithms for Deducing Equalities, Proceedings of the 5th Annual ACM Symposium on Principles of Programming Languages, Tucson, AZ, January 1978, pages 158-170 (also in Computing Reviews 20, 4(April 1979), page 157, entry 34427).

References (Continued)

10. H.Samet, A Canonical Form Algorithm for Proving Equivalences of Conditional Forms, Information Processing Letters 7, 2(February 1978), pages 103-106.

11. H.Samet, Proving Correctness of Heuristically Optimized Code, Communications of the ACM 21, 7(July 1978), pages 570-582.

12. H.Samet, Efficient On-line Proofs of Equalities and Inequalities of Formulas, IEEE Transactions on Computers 29, 1(January 1980), pages 28-32.

13. H.Samet and L.Marcus, Purging in an Equality Data Base, Information Processing Letters 10, 3(March 1980), pages 89-95 (also University of Maryland TR 741).

14. H.Samet, Experience with Software Conversion, Software Practice and Experience 11, 10(1981), pages 1053-1069.

15. H. Samet, Code Optimization Considerations in List Processing Systems, IEEE Transactions in Software Engineering 8, 2(March 1982), pages 107-112.