Specialization of Imperative Programs through Analysis of Relational Expressions

Alexander Sakharov

Motorola
1501 W. Shure Drive
Arlington Heights, IL 60004, USA
sakharov@cig.mot.com
1-708-435-9465

Abstract

An analysis method for specialization of imperative programs is described in this paper. This analysis is an inter-procedural data flow method operating on control flow graphs and collecting information about program expressions. It applies to various procedural languages. The set of analyzed formulas includes equivalences between program expressions and constants, linear-ordering inequalities between program expressions and constants, equalities originating from some program assignments, and atomic constituents of controlling expressions of program branches. Analysis is executed by a worklist-based fixpoint algorithm which interprets conditional branches and ignores some impossible paths. This analysis algorithm incorporates a simple inference procedure that utilizes both positive and negative information. The analysis algorithm is shown to be conservative; its asymptotic time complexity is cubic. A polyvariant specialization of imperative programs, that is based on the information collected by the analysis, is also defined at the level of nodes and edges of control flow graphs. The specialization incorporates a further refinement of analysis information through local propagation. Multiple variants are produced by replicating disjoint subgraphs whose in-links are limited to one node.

Keywords: imperative languages, program specialization, control flow graphs, data flow analysis, fixpoint algorithms, interpretation of conditional branches, polyvariant specialization.

1 Introduction

Despite wide usage of procedural languages, only few successful attempts have been made to apply partial evaluation to these languages. One of the reasons of the limited success of partial evaluation in this domain is due to the fact that advanced data flow analysis techniques created for imperative languages have not been employed to support specialization.

Partial evaluation is called on-line if analysis is done during specialization. Partial evaluation is called off-line if analysis is done before specialization. Off-line techniques, that are considered more appropriate [An, BGZ], employ binding time analysis as the source of information for specialization. Binding time analysis annotates each statement as either static or dynamic by dividing variables into static or dynamic [JGS]. In application to imperative languages, binding time analysis is inferior to sophisticated data flow analysis techniques of compiler optimization technology [ASU].

This paper advocated the use of data flow analysis as the source of information for specialization. An original data flow analysis method that collects information for specialization of imperative programs is presented in this paper. This analysis annotates each program point with a set of static expressions. The analysis reaches far beyond detection of static variables. Expressions whose variables are not static can be classified as static by this analysis.

The control flow graph [ASU] is probably the most adequate, useful, and generic model for analysis and optimization of programs in procedural languages. Data flow analyses operate on environments that are associated with nodes or edges of control flow graphs. In our analysis framework, environments represent relational expressions and other propositions. If analysis of programs in imperative languages is done at the level of control flow graphs, it is natural to also specialize these programs at the same level. An original polyvariant specialization [Bu, JGS] operating on control flow graphs is presented in this paper. Variants are generated by non-recursive code replication.

Analysis of relational expressions and other propositions expands partial evaluation horizons: not only static values of variables but also other assertions may serve as pre-conditions for partial evaluation of imperative programs. In addition to usage for partial evaluation, our analysis and specialization can be viewed as general purpose analysis and optimization methods, respectively.

The following example illustrates our partial evaluation technique. The C code fragment below implements a few transitions of a finite state machine.

```
#define TRANS(A,B,C) { if (state==A) { if (event==B) { state=C;
#define END_TR continue; } } }
...
do { TRANS(off,on_off,on) toggle(); END_TR
if ( vol_trg ) { TRANS(off,vol_up,on) toggle(); volume=1; END_TR }
... } ...
```

If no other transition applies to state off, procedure toggle does not affect variables from this fragment, and vol_trg is a static variable (not equal 0), then the partial evaluation technique presented in this paper makes it possible to transform this loop into more efficient code:

```
do { if (state==off)
     { if (event==on_off)
          { state=on; toggle(); }
     else if (event==vol_up)
```

```
        { state=on; toggle(); volume=1; } }
else ... } ...
```

Here is yet another example (in C) showing capabilities of our partial evaluation technique.

```
int p(int a[], int n, int b) { int s=0; int i=0; int flag=1; float t; float t0=0.01;
do { if ( flag==1 ) { if ( b>=0 ) t=1+2*t0; else t=1+2*v; }
     else if ( a[i]*u>= 0 ) { { if ( a[i]*u+b>=0 ) t=1; else t=1+4*v; } t=(t-0.01)*(1-t0); }
     else t=1+2*v;
     s+=a[i]*t; t0*=0.985; flag=0;
} while ( ++i<=n ); return s; }
```

If n=100 and b≥1 are pre-conditions for partial evaluation of procedure p, then our method in combination with traditional optimizations gives residual code that can be represented in C as follows:

```
int p(int a[], int n, int b) { int s=0; int i=0; int flag=1; float t; float t0=0.01;
t=1.02; s+=a[i]*t; t0*=0.985; ++i;
do { if ( a[i]*u>= 0 ) t=0.99*(1-t0); else t=1+2*v;
s+=a[i]*t; t0*=0.985;
} while ( ++i<=100 ); return s; }
```

The control flow graphs employed here have four types of nodes: assignments, calls, returns, and conditional branches. The environments assigned to nodes and edges are basically conjunctions of certain predicate formulas. These predicate formulas include: atomic constituents of controlling expressions of program branches and their negations; equalities originating from program assignments and equality negations; equivalences between program expressions and constants of the respective types as well as negations of the equivalences; linear-ordering inequalities between program expressions and constants. Other atomic predicate expressions may serve as additional environment constituents. Note that some environment constituents may represent formulas whose predicate symbols or other operations are missing from the language. Binary relations that express aliasing information are an example [LR].

The environments are designed to capture relevant program properties while remaining compact. Note that most predicate symbols in procedural languages are either equivalence relations or linear orderings. Our analysis takes advantage of tracking various properties of all program expressions (and possibly others) simultaneously. Transfer functions [ASU], which are determined by the constructs of the respective language, are not fixed in the framework. Because of this, this framework is applicable to various languages. For instance, the language may include arrays, pointers, and their relevant operations.

The analysis is done by a worklist-based fixpoint algorithm that interprets conditional branches and involves a simple and fast inference procedure. The analysis algorithm presented here is an advanced version of the algorithm from [Sk]. The asymptotic time complexity of the analysis algorithm is proportional to the square of program size times the complexity of the transfer functions in use. Thus, the complexity of this analysis is almost the same as that of conventional constant propagation [WZ].

The inference procedure serves to derive additional propositions. It is repeatedly invoked as propagation progresses. A special treatment of equivalences and linear-ordering inequalities facilitates the inference. The inference procedure exploits both positive atomic propositions and their negations. The inference procedure employs pre-defined evaluation rules. They are determined by language operations and are not fixed in the framework. Though the inference is incomplete, the

inference procedure is an analysis core that distinguishes this analysis algorithm from others. The choice of an inference procedure is driven by the goal to derive as much as possible without imposing a burden in terms of analysis time complexity. Of course, this goal is achieved at expense of theoretical soundness.

Specialization of imperative programs represented as control flow graphs is accomplished through the following steps: disjoint subgraphs whose in-links are limited to one node are selected; selected subgraphs are replicated; local propagation of analysis information is done within subgraphs; static expressions are replaced by the respective constants. The number of specialization variants for a subgraph is limited by the number of its in-links. This non-recursive generation of specialization variants results in smaller size of specialized programs. The specialization process is followed by traditional optimizations on control flow graphs.

Section 2 of this paper outlines control flow graphs in use as well as environments associated with graph elements. Section 3 gives the analysis algorithm and its properties. Program specialization is presented in section 4. Section 5 surveys related work. Pseudo-code of the inference procedure is given in Appendix A. Proofs are given in Appendix B.

2 Analysis Framework

Let us distinguish four types of nodes in the graphs: conditional branches, assignments, calls, and returns. Each conditional branch has two out-edges. The branch node is controlled by a predicate expression. Control is transferred to either the “true” or the “false” edge according to the value of the expression. Assignments embody calculations and variable updates. Each assignment has one out-edge. Each call node also has one out-edge leading to the callee and one out-edge leading to the following node. Return nodes have multiple out-edges leading to all return points. Neither call nor return nodes update any variable values. Calculations of parameter and return values are modeled by assignments in control flow graphs. Zero, one, or several expressions are associated with every node. It is assumed that the size of each of the expressions associated with nodes is bounded by a constant. There are two distinguished dummy nodes: the start node and the end node. The start node has no in-edges and one out-edge. The end node has no out-edges.

We assume that every procedure has one return node. Predicate expressions controlling branches are supposed to be atomic, i.e. they do not contain propositional connectives [CL]. Branches whose controlling expressions contain propositional connectives can be reduced to nested branches without propositional connectives by application of so-called short-circuit rules [FL]. We also assume that assignments do not contain calls and that controlling expressions of conditional branches do not have side effects. Conditional branches with side effects and assignments containing calls can be eliminated through introduction of new variables and inclusion of additional assignments. Note that this transformation of the program control flow graph can be done in linear time.

We do not fix data types and operations permitted in expressions. They depend on the language. For instance, expressions may include array subscripting, operations on pointers, etc. Equivalence relations and linear orderings are preferable for our analysis. When possible, other predicate symbols should be expressed through these. Major constructs of imperative programming languages fall into this control flow graph model. Note that recursive procedures are allowed. Jump tables and dynamic calls/returns are not included in the model for reducing technicality.

Let Θ denote a set of non-predicate expressions without side effects. It includes all arithmetic and language-specific expressions originating from the program and all their subexpressions (except

constants). The cardinal number of Θ should be proportional to program size. Let Π be a set of atomic predicate expressions without side effects. It includes the set of controlling expressions of program branches and equalities originating from program assignments whose left- and right-hand sides belong to Θ and which do not refer to one object in both states: before and after assignment. The cardinal number of Π should be proportional to program size. Although, the only result of not complying with the limitations on the cardinal numbers of Θ and Π is a higher complexity of the analysis.

For the sake of reducing technicality, we do not allow predicate expressions be subexpressions. We assume that one element of Θ or Π corresponds to all textually identical expressions. Moreover, expressions which differ because of inversion of operands of commutative operations are identified, and a single specimen is kept in Θ or Π. This identification can be done by a recursive algorithm operating on pairs of expressions. A more sophisticated identification of equal expressions could be based on further developments of ideas of Knuth and Bendix [KB].

Let us define the domain of environments - Ω. Two special values - undef and none - will be used as values of environment constituents. Every environment V from Ω is triplet (V.s,V.t,V.u). The formulas given by equivalence relations (i.e. reflexive, symmetric, transitive, total relations) connecting expressions from Θ and constants of the respective types are represented by (V.s[1],...,V.s[k]). For every equivalence predicate p from the language and every expression e from Θ, one element of array V.s represents assertions p(e,_), and one element of V.s represents assertions ¬p(e,_). Here and below, _ stands for a constant of the respective type. Every V.s[i] is a constant that takes underscore's position, undef, or none. At minimum, equality is represented by V.s elements.

The formulas given by linear orderings (i.e. reflexive, antisymmetric, transitive, total relations) connecting expressions from Θ and constants are represented by (V.t[1],...,V.t[h]). For every linear ordering p from the language and every expression e from Θ, one element of array V.t represents assertions p(e,_), and one element of V.t represents assertions p(_,e). Every V.t[i] is a constant that takes underscore's position, undef, or none.

The predicate expressions of Π are represented by (V.u[1],...,V.u[m]). One element of V.u corresponds to every predicate expression from Π. Each V.u[i] is undef, none, true, or false.

Environments serve to represent assertions about program points. Every environment V from Ω maps to formula Φ(V) if V does not contain undef:

Φ(V) = Φ(V.s[1]) &...& Φ(V.s[k]) & Φ(V.t[1]) &...& Φ(V.t[h]) & Φ(V.u[1]) &...& Φ(V.u[m])

Φ(V.u[i]) is the respective predicate from Π if V.u[i] is true, its negation if V.u[i] is false, or predicate constant "true" if V.u[i] is none. If V.s[i] representing p(e,_) is constant c, then Φ(V.s[i]) is expression p(e,c). If V.s[i] representing ¬p(e,_) is constant c, then Φ(V.s[i]) is expression ¬p(e,c). If V.s[i] is none, then Φ(V.s[i]) is "true". If V.t[i] representing p(e,_) (p(_,e)) is constant c, then Φ(V.t[i]) is expression p(e,c) (p(c,e)). If V.t[i] is none, Φ(V.t[i]) is "true".

Let us define a binary operation called μ on Ω as follows:

μ(V,V') = (V.s[1]^V'.s[1],...,V.s[k]^V'.s[k],
V.t[1]^V'.t[1],...,V.t[h]^V'.t[h],
V.u[1]^V'.u[1],...,V.u[m]^V'.u[m])

The following rules define ^ for V.s, V.t, and V.u elements:

For any v: undef ^ v = v ^ undef = v; none ^ v = v ^ none = none;

v ^ v = v; true ^ false = false ^ true = none.

Let V.s[i] and V'.s[i] represent p(e,_) (or ¬p(e,_)), and
V.s[i] and V'.s[i] are constants of the respective type:
If p(V.s[i],V'.s[i]) then V.s[i] ^ V'.s[i] = V.s[i], otherwise V.s[i] ^ V'.s[i] = none.

Let V.t[i] and V'.t[i] represent p(_,e), and
V.t[i] and V'.t[i] are constants of the respective type:
If p(V.t[i],V'.t[i]) then V.t[i] ^ V'.t[i] = V.t[i], otherwise V.t[i] ^ V'.t[i] = none.

Let V.t[i] and V'.t[i] represent p(e,_), and
V.t[i] and V'.t[i] are constants of the respective type:
If p(V'.t[i],V.t[i]) then V.t[i] ^ V'.t[i] = V.t[i], otherwise V.t[i] ^ V'.t[i] = none.

Operation μ plays the role of "meet", but it is not commutative. Non-commutativity is due to ^ for V.t elements. This operation is basically widening [CC, Bo]. We will use notation $x \leq y$ if the formula $x=\mu(x,y)$ holds. The same notation will be also used in application to environment constituents: $a \leq b$ iff $a=a \wedge b$.

Proposition 1. Relation ≤ is a partial ordering, i.e. for any x,y,z:

$x \leq x$

$x \leq y \ \& \ y \leq z \Longrightarrow x \leq z$

(Sometimes, antisymmetry is also postulated in definition of partial ordering.)

For any x and y:

$\mu(x,y) \leq x$

$\mu(x,y) \leq y$

Proposition 2. If $x \leq y$, neither x nor y contains undef, then $\Phi(y)$ implies $\Phi(x)$.

It is assumed that transfer function $f_N: \Omega \rightarrow \Omega$ is defined for every assignment node N [ASU, KU, He]. Transfer functions depend on the language in use. In presence of pointers, transfer functions should subsume approximation of alias effects [ASU]. Consider a sample transfer function for assignment v:=e where v is an integer variable and e is an arithmetic expression. First, the transfer function sets to none all V.s, V.t, and V.u elements which contain v. Second, the transfer function sets V.s and V.t elements representing v to the respective elements representing e. Third, if v=e belongs to Π, then the transfer function sets V.u[i] representing v=e to true.

3 Analysis Algorithm

We use a worklist-based fixpoint algorithm (BC) to propagate environments. This algorithm updates the worklist by symbolically interpreting the program. It starts with an optimistic assumption about propagated values and proceeds by changing the values until it reaches a fixed point. BC utilizes algorithm R which does an incomplete inference. Results of this paper apply to any other more advanced inference algorithm while that other algorithm satisfies the statement of Lemma 1 (see below) and while its time complexity is the same, i.e. linear.

Elements of the worklist W employed by the algorithm are pairs: the pair comprises a value from Ω and an edge. Also, nodes are placed on W to serve as marks. The marks enable usage of W as a stack. The value from Ω assigned to node N is denoted AN(N). The value from Ω assigned to edge E is denoted AE(E). The sets AN and AE for nodes and edges of the control flow graph constitute the output of BC.

D(N) denotes the sole descendant of assignment/start/call node N. If N is a branch node, then $D_t(N)$ and $D_f(N)$ denote the "true" and "false" descendants of N, respectively. $D_c(N)$ is the callee for node N, and $D_r(N)$ is the return node of the callee. S is the start node. The transfer function of assignment node N is denoted f_N. Let I stand for the entire program input including initial variable values. A(I) will denote an environment whose constituents are none except for constants representing equalities for static variables or other pre-conditions.

Algorithm R is given in the Appendix. R evaluates expressions from both Θ and Π. It also derives equivalences and their negations, linear-ordering inequalities, and assertions from other assertions, equivalences, and inequalities.

We assume that elements of arrays s and t in environments are ordered by the size of expressions from Θ. Loops in R (including application of evaluation rules) should iterate from smaller expressions to bigger ones. We assume that two tables - one for elements of Θ and one for elements of both Θ and Π - are created before running BC. Each entry of the first table contains indices of the s and t elements that represent properties of the respective expression from Θ. Each entry of the second table contains references to the first table for all immediate subexpressions of the expression from Θ or Π. These tables make it possible to execute R in linear time.

One or several links may be established for an expressions from Π to other expressions from Π before running BC. If p_1 has a link to p_2, then p_1 is equal to p_2 after substituting constants for some of p_1's subexpressions.

Evaluation rules serve to propagate equivalence and linear-ordering formulas connecting expressions and constants over arithmetic and other non-predicate operations of the language. Given these formulas for immediate subexpressions, the evaluation rule postulates another formula for the expression. Evaluation rules may vary for different languages. One class of evaluation rules is common for all languages, though. Any rule of this class applies an arithmetic operation whose arguments equal constants:

$\{e_1=c_1\},\{e_2=c_2\} \Longrightarrow \{e_1 \tau e_2 = c_1 \tau c_2\}$

Here, τ is a binary arithmetic operation. All evaluation rules ought to be valid formulas if the comma is treated as conjunction [CL]. Interval arithmetic gives other examples of evaluation rules. Here is one of them:

$\{e_1 \le c_1\},\{e_2 \le c_2\} \Longrightarrow \{e_1+e_2 \le c_1+c_2\}$

Algorithm BC

```
set AN(S) <- A(I);
set W <- { ( A(I),(S,D(S) ) };
for every AN(N) except AN(S) do
    set AN(N) <- (undef,...,undef);
for every AE(N,M) do
    set AE(N,M) <- (undef,...,undef);
while W is not empty do begin
    if there is pair ( B,(M,N) ) in W after the last mark K then begin
        set AE(M,N) <- μ(AE(M,N),B);
        set AN(N) <- μ(AN(N),AE(M,N));
        if N is an assignment and
        μ(AE(N,D(N)),R(f_N(AE(M,N))),(N,D(N))) is different from AE(N,D(N)) then
```

```
            add ( R(f_N(AE(M,N))),(N,D(N)) ) to W;
        else if N is a call and
        μ(AE(N,D_c(N)),AN(N)) is different from AE(N,D_c(N)) then begin
            put N on W as a mark;
            add ( AE(M,N),(N,D_c(N)) ) to W end
        else if N is a branch controlled by an expression represented by u[i] then
            if AE(M,N).u[i] = true and
            μ(AE(N,D_t(N)),AE(M,N)) is different from AE(N,D_t(N)) then
                add ( AE(M,N),(N,D_t(N)) ) to W
            else if AE(M,N).u[i] = false and
            μ(AE(N,D_f(N)),AE(M,N)) is different from AE(N,D_f(N)) then
                add ( AE(M,N),(N,D_f(N)) ) to W
            else if AE(M,N).u[i] = none then begin
                if μ(AE(N,D_t(N)),R((AE(M,N).s[1],...,true,...,AE(M,N).u[m])))
                is different from AE(N,D_t(N)) then
                    add ( R((AE(M,N).s[1],...,true,...,AE(M,N).u[m])),(N,D_t(N)) ) to W
                if μ(AE(N,D_f(N)),R((AE(M,N).s[1],...,false,...,AE(M,N).u[m])))
                is different from AE(N,D_f(N)) then
                    add ( R((AE(M,N).s[1],...,false,...,AE(M,N).u[m])),(N,D_f(N)) ) to W
                (true and false are the i-th elements of u) end
    end else begin
        remove K from W;
        if μ(AE(D_r(K),D(K)),AN(D_r(K)) is different from AE(D_r(K),D(K)) then
            add ( AN(D_r(K)),(D_r(K),D(K)) ) to W
    end
end
```

Let A(I,S...N) denote an environment whose constituents are true, false, or constants. A(I,S...N) is calculated for actual values of program expressions at the end of execution of path S...N which is exercised for input I. A(I,S...N).u[j] is the boolean value of the respective predicate expression. For s[j] representing p(e,_), A(I,S...N).s[j] is the value of e. For s[j] representing ¬p(e,_), A(I,S...N).s[j] is some value of the respective type: $neq_p(e)$. It is not equivalent (w.r.t. p) to the value of e. For t[j] representing p(e,_) or p(_,e), A(I,S...N).t[j] is the value of expression e.

Let g be the number of edges in the program control flow graph. $\overline{AN}(N)$ and $\overline{AE}(N,M)$ will denote AN(N) and AE(N,M) after BC termination. Let t stand for the maximum time of transfer function execution. Apparently, t is at least O(g). When calculating time complexity, we assume that the time of executing assignments and operations from expressions under consideration is bounded by a constant. Note that this typical assumption about language operations can be relaxed at the cost of a higher analysis complexity.

Theorem 1. Algorithm BC will eventually terminate on any control flow graph. The asymptotic time complexity of BC is $O(g^2*t)$.

Note that the worst case is rarely achieved because AN(N) and AE(M,N) rapidly converge to fixed points (likewise other optimistic propagation algorithms behave). Actual running time is close to the best-case running time. The best-case running time of BC is O(g*t). The asymptotic time complexity of preliminary actions - construction of tables and links - is not higher that BC's asymptotic complexity.

Definition 1. Transfer function f_N is called safe if Φ(A(I,S...N)) ==> Φ(x) implies that

$\Phi(A(I,S...NN")) ==> \Phi(f_N(x))$ for any x, any input I, and any execution path S...NN" resulting from the input.

Lemma 1. $\Phi(x) ==> \Phi(R(x))$ holds for any x without undef.

Theorem 2. If all transfer functions are safe, then all $\overline{AN}$ and $\overline{AE}$ are conservative, that is, $\Phi(A(I,S...N)) ==> \Phi(\overline{AN}(N))$ for any input I and execution path S...N resulting from I, and $\Phi(A(I,S...NN")) ==> \Phi(\overline{AE}(N,N"))$ for any input I and execution path S...NN" resulting from I.

4 Specialization

Specialization is done on the control flow graph representations of imperative programs. It utilizes $\overline{AE}$ and $\overline{AN}$ values. The specialization is accomplished by the following steps.

1. Disjoint subgraphs are segregated. A subgraph is selected for every conditional branch and every assignment which have multiple in-edges. These subgraphs are recursively defined. The above node is the top node of the respective subgraph. All assignments and conditional branches that are immediate successors of any node from the subgraph and that have one in-edge belong to the respective subgraph along with their in-edges.

2. Each selected subgraph is replicated as many times as there are in-edges with different $\overline{AE}$ values for in-edges of its top node before this replication process starts. Then, out-links are added to each replicated copy. These edges lead to the same nodes as the original out-links do. The in-edges of subgraphs are distributed among copies so that all edges with the same $\overline{AE}$ value lead to one copy. (Note that all in-edges which are also out-edges for a different subgraph lead to one copy.)

3. $\overline{AE}$ value for the in-edge of the top node of every subgraph copy is propagated across the subgraph to update $\overline{AE}$ and $\overline{AN}$ values within the subgraph. $\overline{AE}$ and $\overline{AN}$ are set by applying transfer functions and then R to propagate environments across assignments and by setting one .u element to true/false and then applying R to propagate environments across conditional branches. It is similar to what BC does.

4. Expressions from Θ occurring in nodes are replaced by constants if their respective equalities represented by .s elements of $\overline{AN}$ are set to constants. This replacement is executed top-down.

Figure 2 exhibits the result of transforming the control flow graph fragment in Figure 1 in the process of its specialization. Rectangles in figures depict assignments, calls, or their sequences. Triangles depict conditional branches.

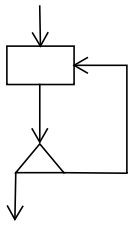

Figure 1

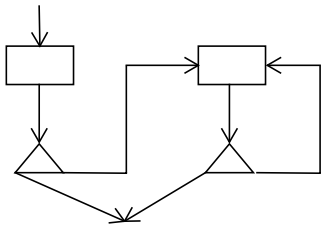

Figure 2

It is assumed that this specialization is followed by traditional optimizations on control flow graphs. These traditional optimizations include: elimination of conditional branches whose con-

trolling expressions are static; mergence of identical nodes whose out-edges lead to the same nodes; elimination of unreachable code; dead-store elimination; loop unrolling; procedure in-lining; copy propagation [ASU]. Note that nodes whose $\overline{AN}$ contain undef and edges whose $\overline{AE}$ contain undef are unreachable.

Theorem 3. $\overline{AN}$ and $\overline{AE}$ remain conservative in the specialized program. The number of nodes in the specialized program is $O(g^2)$ at most.

In practice, the number of nodes in the specialized program is not too big. This happens because the subgraphs are usually small and few edges lead into most subgraphs (see proof of Theorem 3 in Appendix B). Normally, the number of edges leading to a subgraph is reflective of the level of nesting conditional branches and loops. Since the nesting level is normally bounded by a small constant, the number of nodes in specialized programs is linear in practice. Application of the aforementioned traditional optimizations after the specialization would further reduce the size of the specialized program

Let us look at the first example from the Introduction section again. Figure 3 below depicts the control flow graph of this code fragment. The following table gives numbering for nodes in the graph:

1: state==off	2: event==on_off	3: state=on;	4: toggle();
4: vol_trg	5: event==vol_up	6: volume=1;	

Figure 4 illustrates specialization outcome. Figure 5 shows the same code fragment after elimination of static conditional branches (lower 4 and 1 in Figure 4) and elimination of unreachable code (left lower 5 and 3-4-6). Copies of node 1 in the lower part of Figure 2 are static because assertions ¬state==off and state==off are propagated into the left- and right-hand-side subgraphs, respectively, during the specialization. Note that this is a simple example that under-utilizes BC's capabilities. The power of inference procedure R is not used here at all. Evaluation rules and expression links are not needed for this example either.

The second example from the Introduction section exhibits more power of our partial evaluation. In order to get the residual code shown in Introduction, the aforementioned evaluation rules from the Analysis Algorithm section should be utilized. R derives that the proposition b>=0 is always true. The interval arithmetic evaluation rule helps to derive that a[i]*u+b>=0 is always true. The fact that the environment of the loop back edge is propagated to the "false" edge of if (flag==1) only helps to eliminate one check for loop termination. The assignments t=1; and flag=0; are dead after the specialization. The three following code pieces are replicated in the process of specialization:

```
if ( flag==1 ) { if ( b>=0 ) t=1+2*t0; else t=1+2*v; }
else if ( a[i]*u>= 0 ) { { if ( a[i]*u+b>=0 ) t=1; else t=1+4*v; } ... }
else t=1+2*v;
---
t=(t-0.01)*(1-t0);
---
s+=a[i]*t; t0*=0.985; flag=0; } while ( ++i<=n );
```

It is assumed that the following traditional optimizations are applied after the specialization: elimination of conditional branches whose controlling expressions are static; mergence of identical nodes whose out-edges lead to the same nodes; elimination of unreachable code; dead-store elimination.

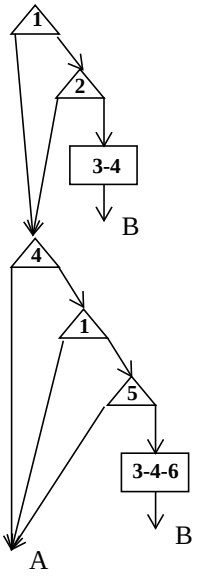

Figure 3

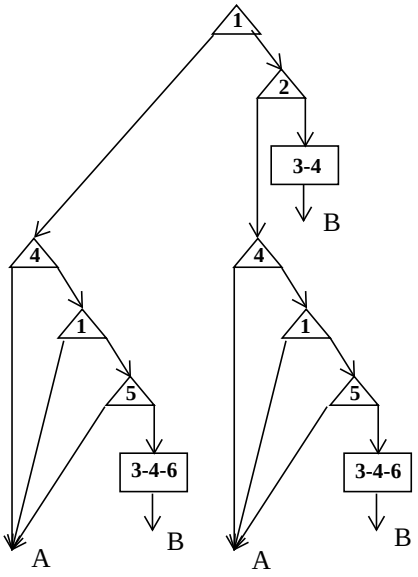

Figure 4

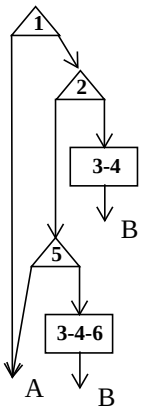

Figure 5

5 Related Work

Meyer studied on-line partial evaluation for a Pascal-like language [Me]. Andersen developed a partial evaluator for a substantial subset of C [An]. Baier, Glueck, and Zochling created a partial evaluator for Fortran [BGZ]. The two latter partial evaluators utilize off-line methods that employ binding time analysis. The analysis of [An] is based on type inference and constraint set solving. The analysis of [BGZ] is a fixpoint iteration over the static/dynamic division of variables. Our analysis iterates over the static/dynamic division not only for variables but also for other program expressions. Moreover, our analysis exploits other relational expressions as an additional source of information for finding static expressions. It incorporates inference capabilities. In contrast to other partial evaluation techniques for imperative languages, our specialization is done at the level of control flow graphs. A different method for limiting the number of specialization variants is applied.

Our framework does not fit the mold of existing static analysis formalisms. It cumulates much more information than binding time analysis [JGS] does. It is not a monotone data flow analysis framework [KU,He] because the monotone framework requires commutative “meet” operation on a semilattice and because interpretation of conditional branches is not expressible in the monotone frameworks. The formalism from [We] is also based on commutative operations. “Meet” of our framework is not commutative. Our framework is not cast in the mold of the classical abstract interpretation [CC] because some controlling expressions of conditional branches may not be specified by binding program variables to values. The environments tracked in our framework are representable in the frameworks from [Sg] (extended to handle conditions) only if all conditions of program branches are expressed by equalities.

The closest analysis frameworks to ours are the constant and assertion propagation framework from [Sk] and the framework utilizing logical expressions whose constituents are equalities about value numbers [Jo]. The class of formulas tracked in the framework of [Sk] is substantially narrower. An inter-procedural analysis is presented here. Algorithm BC ignores impossible pairs in-edge/out-edge for conditional branches. Inference capabilities of the analysis algorithm from [Sk] are much weaker. The framework from [Jo] does not include interpretation of conditional branches and does not incorporate any inference mechanism. The analysis presented here subsumes binding time analysis [JGS, BGZ], conditional constant propagation [WZ, CH], and number interval propagation with widening [CC, Bo] (widening is embedded into "meet").

Our approach to generation of specialization variants results in smaller programs than those produced by the polyvariant program point specialization [Bu, JGS]. The total number of specialization variants for any node is limited by the number of the in-links of its subgraph (as opposed to the number of static memory states). The number of in-links is normally bounded by a small constant.

Analysis and optimization algorithms for elimination of conditional branches in RTL are proposed in [MW]. The analysis algorithm from [MW] operates on basic blocks of a single loop. Once inner loops have been processed, the effects of their blocks are united before processing the outer loop. Thus, that method applies to well-structured programs. The analysis from [MW] incorporates 1-step inference which is substantially weaker than algorithm R. The analysis algorithm from [MW] is a pessimistic algorithm. Generally, pessimistic algorithms are less powerful than optimistic ones. Yet another advantage of BC over the analysis from [MW] is that BC ignores impossible pairs of in/out-edges for conditional branches. The optimization algorithm from [MW], that is based on code replication, may lead to exponential growth of code size. The specialization described in this paper may lead to quadratic code growth at most.

References

[ASU] A.V. Aho, R. Sethi, J. D. Ullman, *Compilers. Principles, Techniques, and Tools*, Addison-Wesley Publ. Co., 1986.

[An] L. O. Andersen. Partial Evaluation of C and Automatic Compiler Generation. *Lecture Notes in Computer Science*, v. 641, 1992, 251-257.

[BGZ] R. Baier, R. Glueck, R. Zochling. Partial Evaluation of Numerical Programs in Fortran. *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation,* 1994.

[Bo] F. Bourdoncle. Abstract Debugging of Higher-Order Imperative Languages. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1993, 47-55.

[Bu] M. A. Boulyonkov. Polyvariant Mixed Computation for Analyzer Programs. *Acta Informatica,* v. 21, 1984, 473-484.

[CH] P.R. Carini, M. Hind. Flow-Sensitive Interprocedural Constant Propagation. *Proceedings of the SIGPLAN Symposium on Programming Language Design and Implementation,* 1995, 23-31.

[CL] C.-L. Chang, R. C.-T. Lee. *Symbolic Logic and Mechanical Theorem Proving*, Academic Press, 1973.

[CC] P. Cousot, R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. *Proceedings of the 4-th ACM Symposium on Principles of Programming Languages*, 1977, 238-252.

[FL] C.N. Fisher, R.J. LeBlanc, Jr. *Crafting A Compiler*, The Benjamin/Cummings Publ. Co., 1988.

[He] M.S. Hecht. *Flow Analysis of Computer Programs*, Elsevier North-Holland, 1977.

[Jo] H. Johnson. Data Flow Analysis for 'Intractable' System Software. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1986, 109-117.

[JGS] N.D. Jones, C. K. Gomard, P. Sestoft. *Partial Evaluation and Automatic Program Generation*, Prentice Hall, 1993.

[KU] J.B. Kam, J.D. Ullman. Monotone Data Flow Analysis Frameworks. *Acta Informatica*, v. 7, 1977, 305-317.

[KB] D.E. Knuth, P.B. Bendix. Simple Word Problems in Universal Algebras. In: Computational Problems in Abstract Algebra (J. Leech, Ed.), Pergamon Press, 1970.

[LR] W. Landi, B.G. Ryder. A Safe Approximate Algorithm for Interprocedural Pointer Aliasing. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1992, 235-248.

[Me] U. Meyer. Techniques for Partial Evaluation of Imperative Languages. *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, 1991, 94-105.

[MW] F. Mueller, D.B. Whalley. Avoiding Conditional Branches by Code Replication. *Proceedings of the SIGPLAN Symposium on Programming Language Design and Implementation*, 1995, 56-66.

[Sg] S. Sagiv et al. A Logic-Based Approach to Data Flow Analysis Problems. *Lecture Notes in Computer Science*, v. 456, 1990, 52-65.

[Sk] A. Sakharov. Propagation of Constants and Assertions. *SIGPLAN Notices*, v. 29, 1994, #5, 3-6.

[We] B. Wegbreit. Property Extraction in Well-Founded Property Sets. *IEEE Transactions on Software Engineering*, 1(1977), #3, 270-285.

[WZ] M.N. Wegman, F.K. Zadeck. Constant Propagation with Conditional Branches. *ACM Transactions on Programming Languages and Systems*, v. 13, 1991, #2, 181-210.

Appendix A

Algorithm R(V)

```
set R(V) <- V;
for every R(V).u[j] which is true and represents p(e1,e2)
where p is an equivalence relation do begin
    if e1 is constant c or R(V).s[i1] representing p(e1,_) is c then
        if R(V).s[i2] representing p(e2,_) is none then set R(V).s[i2] <- c;
    if e2 is constant c or R(V).s[i2] representing p(e2,_) is c then
        if R(V).s[i1] representing p(e1,_) is none then set R(V).s[i1] <- c;
    if R(V).s[i1] representing ¬p(e1,_) is constant c then
        if R(V).s[i2] representing ¬p(e2,_) is none then set R(V).s[i2] <- c;
    if R(V).s[i2] representing ¬p(e2,_) is constant c then
        if R(V).s[i1] representing ¬p(e1,_) is none then set R(V).s[i1] <- c end
INEQ
EVAL
for every R(V).u[j] which is false and represents p(e1,e2)
where p is an equivalence relation do begin
    if e1 is constant c or R(V).s[i1] representing p(e1,_) is c then
        if R(V).s[i2] representing ¬p(e2,_) is none then set R(V).s[i2] <- c;
    if e2 is constant c or R(V).s[i2] representing p(e2,_) is c then
        if R(V).s[i1] representing ¬p(e1,_) is none then set R(V).s[i1] <- c end
EVAL
for every R(V).u[j] which represents p(e1,e2) where p is a linear ordering do begin
    if R(V).u[j] is true, e1 is constant c or R(V).t[i1] representing p(_,e1) is c then
        if R(V).t[i2] representing p(_,e2) is none then set R(V).t[i2] <- c;
    if R(V).u[j] is true, e2 is constant c or R(V).t[i2] representing p(e2,_) is c then
        if R(V).t[i1] representing p(e1,_) is none then set R(V).t[i1] <- c;
    if R(V).u[j] is false, e1 is constant c or R(V).t[i1] representing p(e1,_) is c then
        if R(V).t[i2] representing p(e2,_) is none then set R(V).t[i2] <- c;
    if R(V).u[j] is false, e2 is constant c or R(V).t[i2] representing p(_,e2) is c then
        if R(V).t[i1] representing p(_,e1) is none then set R(V).t[i1] <- c end
EVAL
for every R(V).t[j1] and R(V).t[j2] which represent p(e,_) and p(_,e) respectively do
    if R(V).t[j1] and R(V).t[j2] are constants and R(V).t[j1] = R(V).t[j2] then
        if R(V).s[i] representing e=_ is none then set R(V).s[i] <- R(V).t[j1];
INEQ
EVAL
for every R(V).u[j] which is none and represents p(e1,e2)
where p is an equivalence relation do begin
    if e1 is constant c1 or R(V).s[i1] representing p(e1,_) is c1,
    e2 is constant c2 or R(V).s[i2] representing p(e2,_) is c2 then
        if p(c1,c2) then set R(V).u[j] <- true else set R(V).u[j] <- false;
    if e1 is constant c1 or R(V).s[i1] representing p(e1,_) is c1,
    R(V).s[i2] representing ¬p(e2,_) is constant c2, and p(c1,c2) then
        set R(V).u[j] <- false;
    if R(V).s[i1] representing ¬p(e1,_) is constant c1,
    e2 is constant c2 or R(V).s[i2] representing p(e2,_) is c2, and p(c1,c2) then
        set R(V).u[j] <- false end
```

```
for every R(V).u[j] which is none and represents p(e1,e2)
where p is a linear ordering do begin
    if e1 is constant c1 or R(V).t[i1] representing p(e1,_) is c1,
    e2 is constant c2 or R(V).t[i2] representing p(_,e2) is c2,
    and p(c1,c2) then
        set R(V).u[j] <- true;
    if e1 is constant c1 or R(V).t[i1] representing p(_,e1) is c1,
    e2 is constant c2 or R(V).t[i2] representing p(e2,_) is constant c2,
    p(c2,c1), and c1≠c2 then
        set R(V).u[j] <- false end
for every R(V).u[j] which is none and represents e1=e2 and
for every linear ordering p do begin
    if e1 is constant c1 or R(V).t[i1] representing p(e1,_) is c1,
    e2 is constant c2 or R(V).t[i2] representing p(_,e2) is c2,
    p(c1,c2), and c1≠c2 then
        set R(V).u[j] <- false;
    if e1 is constant c1 or R(V).t[i3] representing p(_,e1) is c1,
    e2 is constant c2 or R(V).t[i4] representing p(e2,_) is c2,
    p(c2,c1), and c1≠c2 then
        set R(V).u[j] <- false end
for every R(V).u[j] which is none and represents e1=e2 and
for every equivalence relation p do begin
    if e1 is constant c1 or R(V).s[i1] representing p(e1,_) is c1,
    e2 is constant c2 or R(V).s[i2] representing p(e2,_) is c2, and ¬p(c1,c2) then
        set R(V).u[j] <- false;
    if e1 is constant c1 or R(V).s[i1] representing p(e1,_) is c1,
    R(V).s[i2] representing ¬p(e2,_) is constant c2, and p(c1,c2) then
        set R(V).u[j] <- false;
    if R(V).s[i1] representing ¬p(e1,_) is constant c1,
    e2 is constant c2 or R(V).s[i2] representing p(e2,_) is c2, and p(c1,c2) then
        set R(V).u[j] <- false end
for every R(V).u[j] which is none and represents p(e1,..,ek) where
e1 is constant c1 or R(V).s[i1] representing e1=_ is c1 ,...,
ek is constant ck or R(V).s[ik] representing ek=_ is ck do
    set R(V).u[j] <- p(c1,...,ck)
for every R(V).u[j1] which is none and represents p1 and for every its link to p2 do
    if R(V).u[j2] representing p2 is true or false and p1 equals p2 after subsituting constants
    for all p1's and p2's subexpressions whose R(V).s[...] representing equality relation
    are constants then
        set R(V).u[j1] <- R(V).u[j2]
```

Here, INEQ is a shorthand notation for:

```
for every R(V).t[j] which is none and represents p(e,_) (or p(_,e)) do
    if R(V).s[i] representing e=_ is constant c then
        set R(V).t[j] <- c;
```

EVAL stands for the following:

```
for every expression from Θ and every relevant evaluation rule do
    apply the rule and update the respective s and t elements if they raise from none;
```

Appendix B

Proposition 1. Relation $\leq$ is a partial ordering, i.e. for any x, y, z:

$x \leq x$

$x \leq y \ \& \ y \leq z \Longrightarrow x \leq z$

For any x and y:

$\mu(x,y) \leq x$

$\mu(x,y) \leq y$

Proof. Reflexivity is obvious. If $x \leq y$ and $y \leq z$, then x.u[i]=x.u[i]^y.u[i] and y.u[i]=y.u[i]^z.u[i] for i=1,...,m. Similar equalities hold for the s and t components of x, y, and z. The fact that the aforementioned equalities imply equalities x.s[i]=x.s[i]^z.s[i], x.t[i]=x.t[i]^z.t[i], x.u[i]=x.u[i]^z.u[i] is proven by considering all possible cases for constituents of x, y, and z: undef, none, true, false for u[i]; undef, none, constants for s[i] and t[i] (including cases with differently related constants).

The inequality $\mu(x,y) \leq x$ is proven by checking that the following equalities hold for any x and y: (x.u[i]^y.u[i])^x.u[i] = x.u[i]^y.u[i]; (x.t[i]^y.t[i])^x.t[i] = x.t[i]^y.t[i]; (x.s[i]^y.s[i])^x.s[i] = x.s[i]^y.s[i]. The inequality $\mu(x,y) \leq y$ is proven by the same method.

Proposition 2. If $x \leq y$, neither x nor y contains undef, then $\Phi(y)$ implies $\Phi(x)$.

Proof. Clearly, the inequality $x.s[i] \leq y.s[i]$ implies that $\Phi(y.s[i]) \Longrightarrow \Phi(x.s[i])$. Similar implications hold for pairs (x.t[i],y.t[i]) and (x.u[i],y.u[i]). Therefore, $\Phi(y.s[1]) \ \& ... \ \& \ \Phi(y.s[k]) \ \& \ \Phi(y.t[1]) \ \& ... \ \& \ \Phi(y.t[h]) \ \& \ \Phi(y.u[1]) \ \& ... \ \& \ \Phi(y.u[m])$ implies $\Phi(x.s[1]) \ \& ... \ \& \ \Phi(x.s[k]) \ \& \ \Phi(x.t[1]) \ \& ... \ \& \ \Phi(x.t[h]) \ \& \ \Phi(x.u[1]) \ \& ... \ \& \ \Phi(x.u[m])$ if $x \leq y$.

Theorem 1. Algorithm BC will eventually terminate on any program flow graph. The asymptotic time complexity of BC is $O(g^2*t)$.

Proof. The time to execute initialization steps of BC is proportional to the total number of nodes and edges in the program graph. Algorithm BC terminates when worklist W is empty. At most two new elements can be added to W at each iteration of BC's main loop. This happen only when AE(E) changes. AE(E).u[i] can only drop twice: from undef to true or false and then to none. Similarly, AE(E).s[i] and AE(E).t[i] can drop twice at most: from undef to a constant, and then to none. Values k, l, and m are proportional to g. Hence, the main loop may add $O(g^2)$ elements to W at most. Therefore, BC will terminate.

The time of executing operation μ and that of assigning an environment to AN(N) or AE(N) is proportional to g. Hence the running time of the initialization phase of BC is proportional to g^2. The time complexity of algorithm R is O(g) because the time of a single execution of the body of each loop in R is bounded by a constant if the two tables are utilized. Since t is at least O(g), the running time of a single iteration of the main loop of BC is proportional to t. The overall time of executing the main loop of BC is proportional to g^2*t. The asymptotic time complexity of BC is $O(g^2*t)$.

Lemma 1. $\Phi(x) \Longrightarrow \Phi(R(x))$ holds for any x without undef.

Proof. If x.u[i] is none and R(x).u[i] is set up to true or false by R, then $\Phi(x) \Longrightarrow \Phi(R(x).u[i])$ holds. If s[i] is raised to a constant from none by R, then $\Phi(x) \Longrightarrow \Phi(R(x).s[i])$ holds. Similarly, if t[i] is raised to a constant from none by R, then $\Phi(x) \Longrightarrow \Phi(R(x).t[i])$ holds. If s[i] or t[i] is raised by an evaluation rule, then the above implication is guaranteed by validity of evaluation rules. Therefore, $\Phi(x) \Longrightarrow \Phi(R(x))$ holds because $\Phi(R(x))$ is conjunction of $\Phi(x)$ and formulas which are implications of $\Phi(x)$.

Theorem 2. If all transfer functions are safe, then all $\overline{AN}$ and $\overline{AE}$ are conservative, that is, $\Phi(A(I,S...N)) ==> \Phi(\overline{AN}(N))$ for any input I and execution path S...N resulting from I, and $\Phi(A(I,S...NN")) ==> \Phi(\overline{AE}(N,N"))$ for any input I and execution path S...NN" resulting from I.

Proof. Suppose to the contrary. Consider a shortest execution path S...N such that: implications $\Phi(A(I,S...M)) ==> \Phi(\overline{AN}(M))$ and $\Phi(A(I,S...M'M)) ==> \Phi(\overline{AE}(M',M))$ hold for any node M on path S...N except N; either one or both of these implications are not valid formulas for N. N is not the start node because $\overline{AN}(S) = A(I)$ and $A(I) \leq A(I,S)$.

Suppose S...N has an edge which has never belonged to any pair from W. Let (M,M") be the first such edge on S...N. M is not the start node because (S,D(S)) is placed on W. Let M' be the predecessor of M on S...N. By our assumption, $\Phi(A(I,S...M'M))$ implies $\Phi(\overline{AE}(M',M))$. Thus $\overline{AE}(M',M)$ does not contain undef. If M is an assignment node, then a pair containing M's out-edge is placed on W each time when AE(M',M) changes. AE(M',M) changes at least once because $\overline{AE}(M',M)$ does not contain undef. In particular, a pair containing (M,M") is placed on W when $\overline{AE}(M',M)$ is attained. Similarly, if M is a call node, then a pair containing (M,M"), where M" is $D_c(M)$, is placed on W when $\overline{AE}(M',M)$ is attained. If M is a return node, then a pair containing (M,M") is placed on W when a mark in W is reached after attaining $\overline{AE}(M',M)$.

$\Phi(A(I,S...M))$ is a consistent formula [CL]: it is true for the values obtained after execution of S...M on input I. If M is a branch controlled by the expression represented by u[i], then $\overline{AE}(M',M).u[i]$ is either equal to A(I,S...M).u[i] or none. In both cases, a pair containing (M,M") is placed on W when $\overline{AE}(M',M)$ is attained. Hence every edge (M,M") from S...N belongs to a pair which is placed on W after $\overline{AE}(M',M)$ is attained.

Let N' stand for the predecessor of N on S...N. If N' is the start node, then $\overline{AE}(S,N) = A(I)$ and $\overline{AN}(N) \leq A(I)$. Hence both $\Phi(A(I,SN)) ==> \Phi(\overline{AN}(N))$ and $\Phi(A(I,SN)) ==> \Phi(\overline{AE}(S,N))$ hold, which contradicts our assumption. Let N" be the predecessor of N' on S...N. By our assumption, $\Phi(A(I,S...N"N'))$ implies $\Phi(\overline{AE}(N",N'))$. If N' is a call or return node, then A(I,S...N"N')=A(I,S...N"N'N). If N' is a call node, then $(\overline{AE}(N",N'),(N'N))$ has been placed on W. If N' is a return node, then $(\overline{AN'}(N'),(N'N))$ has been placed on W, and $\overline{AN'}(N') \leq \overline{AE}(N",N')$ by Proposition 1. If N' is an assignment node, then $(R(f_{N'},(\overline{AE}(N",N'))),(N',N))$ has been on W, and $\Phi(A(I,S...N'N)) ==> \Phi(f_{N'},(\overline{AE}(N",N')))$ since all transfer functions are safe. By Lemma 1, $\Phi(f_{N'},(\overline{AE}(N",N')))$ implies $\Phi(R(f_{N'},(\overline{AE}(N",N'))))$.

If N' is a branch node, let u[i] represent the controlling expression of N'. If (N',N) is the "true" out-edge of N', then either $(\overline{AE}(N",N'),(N',N))$ or $(R((\overline{AE}(N",N').s[1],...,true,...,\overline{AE}(N",N').u[m])), (N',N))$ has been on W. $\overline{AE}(N",N').u[i]$ is either true or none since $\Phi(A(I,S...N'))$ is a consistent formula. Note that $\Phi(A(I,S...N'N)) = \Phi(A(I,S...N'))$. Henceforth, implications $\Phi(A(I,S...N)) ==> \Phi(\overline{AE}(N",N'))$ and $\Phi(A(I,S...N)) ==> \Phi((\overline{AE}(N",N').s[1],...,true,...,\overline{AE}(N",N').u[m]))$ hold. By Lemma 1, $\Phi((\overline{AE}(N",N').s[1],...,true,...,\overline{AE}(N",N').u[m]))$ implies $\Phi(R((\overline{AE}(N",N').s[1],...,true,...,\overline{AE}(N",N').u[m])))$.

If (N',N) is the "false" out-edge, then either $(\overline{AE}(N",N'),(N',N))$ or $(R((\overline{AE}(N",N').s[1],...,false,...,\overline{AE}(N",N').u[m])),(N',N))$ has been on W. $\overline{AE}(N",N').u[i]$ is either false or none. Again, implications $\Phi(A(I,S...N)) ==> \Phi(\overline{AE}(N",N'))$ and $\Phi(A(I,S...N)) ==> \Phi((\overline{AE}(N",N').s[1],...,false,...,\overline{AE}(N",N').u[m]))$ hold. By Lemma 1, $\Phi((\overline{AE}(N",N').s[1],...,false,...,\overline{AE}(N",N').u[m]))$ implies $\Phi(R((\overline{AE}(N",N').s[1],...,false,...,\overline{AE}(N",N').u[m])))$.

In all cases, such (V,(N',N)) has been on W that Φ(A(I,S...N)) ==> Φ(V). The second statement of Proposition 1 guarantees that successive values AN(M) and AE(E) form decreasing sequences for any given node M and edge E. Proposition 1 also guarantees that $\overline{AN}$(N)≤$\overline{AE}$(N',N). Thus, $\overline{AE}$(N',N)≤V and $\overline{AN}$(N)≤V. By proposition 2, Φ(V) implies Φ($\overline{AN}$(N)) and Φ($\overline{AE}$(N',N)). This contradicts the assumption.

Theorem 3. $\overline{AN}$ and $\overline{AE}$ remain conservative in the specialized program. The number of nodes in the specialized program is $O(g^2)$ at most.

Proof. $\overline{AN}$ values for calls, returns, and other nodes with multiple in-edges in the source program and $\overline{AE}$ values for edges leading to the above nodes remain unchanged in the specialized program. Consider a shortest path S...N such that $\overline{AN}$ and $\overline{AE}$ are conservative for all nodes on the path but N. N should be an internal node of one of the replicated subgraphs, i.e. it is either an assignment or a conditional branch with one in-edge, because the path is the same as it would be in the source program, and $\overline{AN}$ and $\overline{AE}$ are not changed for other nodes.

Let N' be the immediate predecessor of N on the path, and let N" be the immediate predecessor of N'. By the assumption, Φ(A(I,S...N"N')) ==> Φ($\overline{AE}$(N",N')). If N' is an assignment, then $\overline{AE}$(N',N)=$\overline{AN}$(N)=R(f_N,($\overline{AE}$(N",N')). If N' is a conditional branch and (N',N) is the "true" edge, then $\overline{AE}$(N',N)=$\overline{AN}$(N)=R(($\overline{AE}$(N",N').s[1],...,true,...,$\overline{AE}$(N",N').u[m])). If (N',N) is the "false" edge, then $\overline{AE}$(N',N)=$\overline{AN}$(N)=R(($\overline{AE}$(N",N').s[1],...,false,...,$\overline{AE}$(N",N').u[m])). The above equalities hold because (N',N) is the only in-edge of N. Now, Theorem 2 reasoning applies to derive the following: Φ(A(I,S...N'N)) ==> Φ($\overline{AE}$(N',N)); Φ(A(I,S...N'N)) ==> Φ($\overline{AN}$(N)). This contradicts our assumption.

Let e_i be the number of in-edges for the top node of the i-th subgraph, and n_i be the number of nodes in this subgraph. The number of added nodes is less than $e_1*n_1+...+e_r*n_r$, where r is the number of the replicated subgraphs. Apparently, the following inequalities hold: $e_1+...+e_r≤g$; $n_1+...+n_r≤g$. Since $e_1*n_1+...+e_r*n_r≤(e_1+...+e_r)*(n_1+...+n_r)$, the number of added nodes is not more than g^2.