# 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

**Abstract**

An inter-procedural data flow analysis operating on control flow graphs and collecting information about program expressions is described in this paper. The following relational and other expressions are analyzed: equivalences between program expressions and constants; linear-ordering inequalities between program expressions and constants; equalities originating from some program assignments; atomic constituents of controlling expressions of program branches. Analysis is executed by a worklist-based fixpoint algorithm which interprets conditional branches and incorporates a rule-based inference procedure. Two variants of the polyvariant program point specialization using results of the analysis are presented. The both specializations are done at the level of control flow graphs. The variants differ in terms of the size of residual programs.

## 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 used 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 for procedural languages [3], employ binding time analysis to guide specialization. Binding time analysis annotates each statement as either static or dynamic by dividing variables into static or dynamic [17].

This paper presents an off-line technique. It advocated the use of data flow analysis as an information source for specialization. An original data flow analysis method that collects information for specialization of imperative programs is described 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 [1] is probably the most adequate, useful, and generic model for analysis and optimization of programs in procedural languages. Data flow analyses operate on environments [10, 23] that are associated with nodes or edges of control flow graphs. In our analysis framework, environments represent relational expressions and other propositions. Specialization is done at the level of control flow graphs. Two specializations using the information collected by our analysis are presented. One is a variant of the polyvariant program point specialization [6, 17, 3]. This specialization may generate huge residual programs. The other is an original technique which is a limited polyvariant specialization. It results in smaller residual programs. This paper focuses on the limited variant.

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 (see also [12,9]). Our analysis and edge-based specialization can also be viewed as general purpose analysis and optimization methods, respectively.

Environments in our analysis 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 some 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 [20].

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 [1], 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 [24]. 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 [26].

The inference procedure serves to derive additional assertions. It is repeatedly invoked as propagation progresses. The inference procedure is based on rules. Some rules express properties of equivalence relations and linear orderings. Other rules are determined by language operations and are not fixed in the framework. Through the rules, the inference procedure exploits both positive atomic propositions and their negations. Though the inference is incomplete, this heuristic 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.

The complete polyvariant specialization using our analysis will be called *store-based* because program point versions are defined by store sets [16]. Analysis information in the form of environments is used to represent store sets. The limited variant is called *edge-based* because it employes edges of control flow graphs as program point versions. The edge-based specialization 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; other optimizations are performed. The edge-based specialization results in smaller residual code because the number of specialization variants for a subgraph is limited by the number of its in-links.

Section 2 of this paper gives motivating examples which are revisited in section 9. Section 3 outlines control flow graphs in use. Section 4 describes environments associated with graph elements. The rules used in analysis are described in section 5. Section 6 gives the analysis algorithm, while section 7 outlines its properties. Program specialization is presented in section 8. Section 10 surveys related work.

## 2 Motivation

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 edge-based specialization 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 the power 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>= b ) { { if ( a[i]*u+b>=2 ) 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 the edge-based specialization in combination with dead-store elimination [1] gives residual code that can be represented in C as follows:

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

The store-based specialization followed by dead-store elimination results in the code below:

```
int p(int a[], int b) { int s=0; int i=0; int flag=1; float t; float t0=0.01;
s+=a[0]*1.02;
if ( a[1]*u>= b ) t=0.980248; else t=1+2*v;
s+=a[1]*t;
...
if ( a[100]*u>= b ) t=0.987849; else t=1+2*v;
s+=a[100]*t;
return s; }
```

## 3 Program Model

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 then-branch or the else-branch 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. 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. 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 [8]. Branches whose controlling expressions contain propositional connectives can be reduced to nested branches without propositional connectives by application of so-called short-circuit rules [11]. It is assumed that the size of all expressions from conditional branches and assignments is bounded by a constant. 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 the above transformations 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.

## 4 Analysis Framework

Let $\Theta$ 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 $\Theta$ should be proportional to program size. Let $\Pi$ 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 $\Theta$ and which do not refer to one object in both states: before and after assignment. The cardinal number of $\Pi$ should be proportional to program size. Although, the only result of not complying with the limitations on the cardinal numbers of $\Theta$ and $\Pi$ 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 [19].

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 relation 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:

$$\Phi(V)=\Phi(V.s[1])\&...\&\Phi(V.s[k])\&\Phi(V.t[1])\&...\&\Phi(V.t[h])\&\Phi(V.u[1])\&...\&\Phi(V.u[m])$$

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 proposition T if V.u[i] is none [8]. 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 proposition T. 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 T.

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

$$\mu(V,V')=(V.s[1]\wedge V'.s[1],...,V.s[k]\wedge V'.s[k],V.t[1]\wedge V'.t[1],...,V.t[h]\wedge V'.t[h],\\ V.u[1]\wedge V'.u[1],...,V.u[m]\wedge V'.u[m])$$

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 $\mu$ plays the role of meet [18,14], but it is not commutative. Non-commutativity is due to ^ for V.t elements. This operation for V.t elements is similar to widening [10, 5]. 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^b.

**Proposition 1.** Relation $\leq$ is a pre-ordering, i.e. for any x,y,z:
$x \leq x$
$x \leq y$ & $y \leq z$ ==> $x \leq z$

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 .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]) ==> $\Phi$(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 $x \leq y$. ■

It is assumed that transfer function $f_N$: $\Omega$->$\Omega$ is defined for every assignment node N [1, 18, 14]. Transfer functions depend on the language in use. In presence of pointers, transfer functions should subsume approximation of alias effects [1]. 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 $\Pi$, then the transfer function sets V.u[i] representing v=e to true.

## 5 Rules

The rules used by the inference procedure have the form of implications. Their consequent is an atomic predicate formula or its negation. The antecedent is conjunction of atomic predicate formulas or their negations. In each rule, there is one selected atomic predicate formula that is called *base*. The base is a pattern for an assertion represented by an environment constituent. All other atomic predicate formulas either express conditions or are patterns for relational expressions represented by .s or .t environment constituents. There are two types of variables in the formulas. Variables of the first type occur in patterns and serve as counterparts of expressions from Θ. All variables of the first type should occur in a base part that is a counterpart of an expression from Θ or Π. Variables of the second type are bound with constants. At least one occurrence of this variable in a rule is a constant counterpart in a formula that is a pattern.

All rules ought to be valid formulas [8]. Rules may vary for different languages. There is a common class of rules, though. It includes the following rules given for all arithmetic (f) and predicate (r) operations:

$e_1=c_1$ & ... & $e_s=c_s$ ==> $f(e_1,...,e_s)= f(c_1,...,c_s)$
$e_1=c_1$ & ... & $e_s=c_s$ & $r(c_1,...,c_s)$ ==> $r(e_1,...,e_s)$
$e_1=c_1$ & ... & $e_s=c_s$ & $\neg r(c_1,...,c_s)$ ==> $\neg r(e_1,...,e_s)$

Other common rules embody properties of equivalence relations and linear orderings. Here are several sample rules from this class:

$\neg p(e_1,e_2)$ & $p(e_1,c_1)$ ==> $\neg p(e_2,c_1)$
$q(e_1,c_1)$ ==> $q(e_1,c_1)$
$q(e_1,e_2)$ & $q(e_2,c_1)$ ==> $q(e_1,c_1)$
$e_1=c_1$ ==> $q(e_1,c_1)$
$p(e_1,c_1)$ & $p(e_2,c_2)$ & $p(c_1,c_2)$ ==> $p(e_1,e_2)$
$p(e_1,c_1)$ & $\neg p(e_2,c_2)$ & $p(c_1,c_2)$ ==> $\neg p(e_1,e_2)$
$q(e_1,c_1)$ & $q(c_2,e_2)$ & $q(c_1,c_2)$ ==> $q(e_1,e_2)$
$q(e_1,c_1)$ & $q(c_1,c_2)$ ==> $q(e_1,c_2)$
$q(e_1,c_1)$ & $q(c_2,e_2)$ & $\neg q(c_1,c_2)$ & $c_1 \neq c_2$ ==> $e_1 \neq e_2$

where p is an equivalence relation, q is a linear ordering, $e_i$ are variables of the first type whereas $c_i$ stand for variables of the second type. Base formulas are shown in italics. The examples below exhibit language-specific rules:

$e_1 \geq c_1$ & $e_2 \geq c_2$ ==> $e_1+e_2 \geq c_1+c_2$
$q(e_1,c_1,)$ & $e_2=1$ ==> $q(e_1*e_2,c_1)$
$e_1-e_2 \leq c_1$ & $e_2 \leq c_2$ ==> $e_1 \leq c_1+c_2$

## 6 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. R evaluates expressions from both Θ and Π. It also derives assertions including equivalences, their negations, and linear-ordering inequalities from other assertions. 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). AN and AE 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 then-descendant and else-descendant 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.

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 given static variables or other pre-conditions.

We assume that elements of arrays .s and .t in environments are ordered by the size of expressions of Θ. Loops in R 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 .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.

**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
        remove (B,(M,N)) from W;
        set AE(M,N) <- $\mu$(AE(M,N),B);
        set AN(N) <- $\mu$(AN(N),AE(M,N));
        if N is an assignment and
        $\mu$(AE(N,D(N)),R($f_N$(AE(M,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
        $\mu$(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
            $\mu$(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
            $\mu$(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 $\mu$(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 $\mu$(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 $\mu$(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
```

**Algorithm R(V)**

```
set R(V) <- V;
for every R(V).u[j] which is true or false and
every evaluation rule with a matching base expression in the antecedent do
    begin match other patterns against relational expressions; check conditions;
    if the antecedent is satisfied then
        update R(V).s[i]/R(V).s[i] represented by a consequent part if it was none end
for every R(V).s[j]/R(V).t[j] which is none and every evaluation rule
whose base part in the consequent matches the respective expression from $\Theta$ do
    begin match other patterns against relational expressions; check conditions;
    if the antecedent is satisfied then
        update R(V).s[j]/R(V).t[j] end
for every R(V).u[j] which is none and
every evaluation rule with a matching base in the consequent do
    begin match other patterns against relational expressions; check conditions;
    if the antecedent is satisfied then
        update R(V).u[j]] end
```

## 7 Analysis Properties

Let A(I,S...N) denote an environment whose constituents are true, false, or constants. A(I,S...N) is calculated for the 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 $O(g^2*t)$.

**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 $\mu$ 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 $O(g^2*t)$. ■

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 usually 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 including construction of tables is not higher than BC's asymptotic complexity.

**Definition.** Transfer function $f_N$ is called *safe* if Φ(A(I,S...N)) ==> Φ(x) implies that Φ(A(I,S...NN")) ==> Φ($f_N$(x)) for any x, any input I, and any execution path S...NN" resulting from the input.

**Lemma 1.** Φ(x) ==> Φ(R(x)) holds for any x without undef.

**Proof.** Validity of the rules implies the following statements about R(x) constituents. If x.u[i] is none and R(x).u[i] is set up to true or false by R, then Φ(x) ==> Φ(R(x).u[i]) holds. If s[i] is raised to a constant from none by R, then Φ(x) ==> Φ(R(x).s[i]) holds. Similarly, if t[i] is raised to a constant from none by R, then Φ(x) ==> Φ(R(x).t[i]) holds. Therefore, Φ(x) ==> Φ(R(x)) holds because Φ(R(x)) is conjunction of Φ(x) and formulas which are implications of Φ(x). ■

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

Φ(A(I,S...M)) is a consistent formula [8]: 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)≤A(I). Hence both Φ(A(I,SN)) ==> Φ($\overline{AN}$(N)) and Φ(A(I,SN)) ==> Φ($\overline{AE}$(S,N)) hold, which contradicts our assumption. Let N" be the predecessor of N' on S...N. By our assumption, Φ(A(I,S...N"N')) implies Φ($\overline{AE}$(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’)≤$\overline{AE}$(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 Φ(A(I,S...N’N)) ==> Φ($f_{N’}$($\overline{AE}$(N”,N’))) since all transfer functions are safe. By Lemma 1, Φ($f_{N’}$($\overline{AE}$(N”,N’))) implies Φ(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 then-edge of N’, then either (R(($\overline{AE}$(N”,N’).s[1],...,true,...,$\overline{AE}$(N”,N’).u[m])),(N’,N)) or ($\overline{AE}$(N”,N’),(N’,N)) has been on W. $\overline{AE}$(N”,N’).u[i] is either true or none since Φ(A(I,S...N’)) is a consistent formula. Note that Φ(A(I,S...N’N)) = Φ(A(I,S...N’)). Henceforth, implications Φ(A(I,S...N)) ==> Φ($\overline{AE}$(N”,N’)) and Φ(A(I,S...N)) ==> Φ(($\overline{AE}$(N”,N’).s[1],...,true,...,$\overline{AE}$(N”,N’).u[m])) hold. By Lemma 1, Φ(($\overline{AE}$(N”,N’).s[1],...,true,...,$\overline{AE}$(N”,N’).u[m])) implies Φ(R(($\overline{AE}$(N”,N’).s[1],...,true,..., $\overline{AE}$(N”,N’).u[m]))).

If (N’,N) is the else-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 Φ(A(I,S...N)) ==> Φ($\overline{AE}$(N”,N’)) and Φ(A(I,S...N)) ==> Φ(($\overline{AE}$(N”,N’).s[1],...,false,...,$\overline{AE}$(N”,N’).u[m])) hold. By Lemma 1, Φ(($\overline{AE}$(N”,N’).s[1],...,false,...,$\overline{AE}$(N”,N’).u[m])) implies Φ(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. ■

## 8 Specialization

Consider the store-based specialization first. Polyvariant specialization techniques select program variables whose calculations can be done at compile time. Their values are computed for every program point in a driven program [16]. Thus, values of these variables belong to finite sets. Normally, loop indices are such variables. Suppose a technique for detection of these variables is selected. Let Λ be the set of the variables. Let store sets be given by $\underline{AE}$(M,N): $\underline{AE}$(M,N) represents all the stores that satisfy Φ($\underline{AE}$(M,N)). $\underline{AE}$(M,N) is obtained from $\overline{AE}$(M,N) in the two following steps. First, environment elements representing v=_ for variables from Λ are changed to the respective constants if the elements were none. Second, R is applied.

Each edge in the control flow graph is a specialization point. Edge (M,N) is specialized with respect to different environments $\underline{AE}$(M,N). Apparently, there are finitely many different $\underline{AE}$(M,N) for every edge (M,N). The store-based specialization runs through the control flow graph, calculates $\underline{AE}$ for current values of variables from Λ, changes expressions to constants on the basis of equalities represented by .s elements of $\underline{AE}$, executes static calculations, and generates code for dynamic nodes.

Polyvariant specializations may generate huge residual programs due to loop unrolling. When code size is a concern, the edge-based specialization below is a preference. The edge-based specialization utilizes $\overline{AE}$ and $\overline{AN}$ values. It 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. 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 are assigned to the respective subgraph along with their in-edges.

2. Each selected subgraph is replicated as many times as there were top node in-edges with different $\overline{\text{AE}}$ not containing undef before this replication process started. 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 $\overline{\text{AE}}$ lead to one copy. Edges whose $\overline{\text{AE}}$ contain undef are eliminated. Note that all in-edges which are also out-edges for a different subgraph lead to one copy.

3. $\overline{\text{AE}}$ of the in-edges of the top node of every subgraph copy is propagated across the subgraph to update $\overline{\text{AE}}$ and $\overline{\text{AN}}$ values within the subgraph. $\overline{\text{AE}}$ and $\overline{\text{AN}}$ 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 $\overline{\text{AN}}$ are set to constants. Conditional branches whose controlling expressions are static are eliminated. Identical nodes whose out-edges lead to the same nodes are merged. Unreachable code, i.e. the nodes whose $\overline{\text{AN}}$ contain undef and the edges whose $\overline{\text{AE}}$ contain undef, is eliminated.

It is assumed that the both specializations are followed by dead-store elimination [1]. Figure 2 exhibits the result of transformation of the control flow graph fragment in Figure 1 in the process of the edge-based specialization. Rectangles in figures depict assignments, calls, or their sequences. Triangles depict conditional branches.

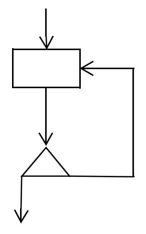
Figure 1

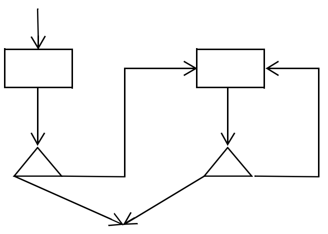
Figure 2

**Theorem 3.** $\overline{\text{AN}}$ and $\overline{\text{AE}}$ remain conservative after the edge-based specialization changes them. The number of nodes in the specialized program is O($g^2$) at most.

**Proof.** $\overline{\text{AN}}$ values for calls, returns, and other nodes with multiple in-edges in the source program and $\overline{\text{AE}}$ values for edges leading to the above nodes remain unchanged in the specialized program. Consider a shortest path S...N such that $\overline{\text{AN}}$ and $\overline{\text{AE}}$ are conservative for all nodes on the path but N. N should be an internal node of one of 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 $\overline{\text{AN}}$ and $\overline{\text{AE}}$ are not changed for the 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($\overline{\text{AE}}$(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 then-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 else-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: $\Phi(A(I,S...N'N)) ==> \Phi(\overline{AE}(N',N))$; $\Phi(A(I,S...N'N)) ==> \Phi(\overline{AN}(N))$. 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 fewer than $e_1*n_1+...+e_r*n_r$, where r is the number of replicated subgraphs. Apparently, the following inequalities hold: $e_1+...+e_r \leq g$; $n_1+...+n_r \leq g$. Since $e_1*n_1+...+e_r*n_r \leq (e_1+...+e_r)*(n_1+...+n_r)$, the number of added nodes is not more than $g^2$. ■

In practice, the number of nodes in the residual program is not big. This happens because the subgraphs are disjoint and few edges lead into most subgraphs (see the proof above). 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, code size growth due to the edge-based specialization is linear in practice. Dead code elimination further reduce code size. Examples from this paper exhibit residual code that is even smaller than the source.

## 9 Examples Revisited

Let us look at the two examples from the Motivation section again. Figure 3 below depicts the control flow graph of the first code fragment. The then-branches are to the right in the figures. The following table gives numbering for nodes in the graph:

| | | | |
|---|---|---|---|
| 1: state==off | 2: event==on_off | 3: state=on; | 4: toggle(); |
| 5: vol_trg | 6: event==vol_up | 7: volume=1; | |

Figure 4 illustrates the outcome of subgraph replication. Figure 5 shows the same code fragment after elimination of static conditional branches (lower 5 and 1 in Figure 4) and elimination of unreachable code (left lower 6 and 3-4-7). Copies of node 1 in the lower side of Figure 4 are static because assertions ¬state==off and state==off are propagated into the left and right copies, respectively, during the edge-based 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.

The second example exhibits more power of our partial evaluation. The rules presented earlier help to derive that the controlling expressions b>=0 and a[i]*u+b>=2 are always true. Note that AE of the loop back edge is propagated solely to the else-branch of if ( flag==1 ). The three following subgraphs are detected and handled by the edge-based specialization:

| | |
|---|---|
| • if ( flag==1 ) ... else t=1+2*v; | (2 copies) |
| • t=(t-0.01)*(1-t0); | (1 copy) |
| • s+=a[i]*t; ... while ( ++i<=n ); | (3 copies, 2 merged) |

If a polyvariant specialization does not use results of our analysis, then loop body instances in the respective residual code could be represented by the following pattern:

```
if ( a[k]*u>= b ) { { if ( a[k]*u+b>=2 ) t=1; else t=1+4*v; } t=(t-0.01)*c; } else t=1+2*v;
s+=a[k]*t;
```

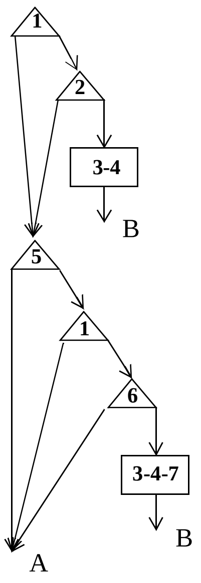

Figure 3

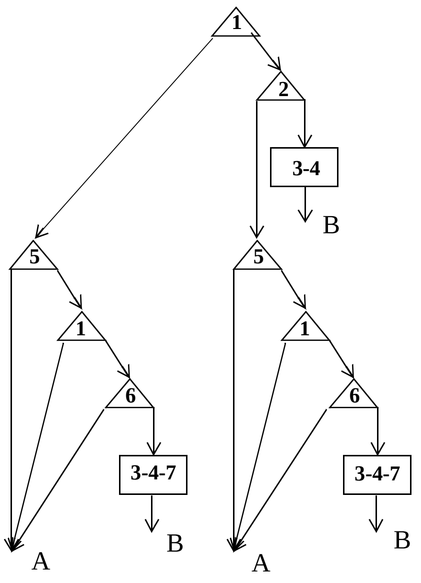

Figure 4

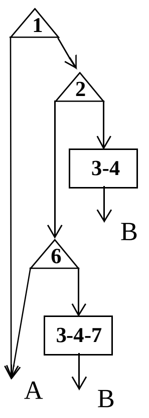

Figure 5

## 10 Related Work

Our analysis cumulates much more information than binding time analysis [17] does. It iterates over the static/dynamic division not only for variables but also for other program expressions. Moreover, our analysis exploits other relational expressions and controlling expressions of conditional branches as another source of information for finding static expressions. The inference procedure derives additional static information from that other source. Hence, more precise information is yielded by our analysis algorithm. Our analysis gives static information per program control flow edge, i.e. several classifications may be given for one node.

Blazy and Facon use constant propagation as a specialization enabler [4]. If a controlling expressions of a conditional branch reduces to an equality between a variable and a constant, then their method propagates the constant in the then-branch. Glueck and Klimov [13] developed a method for propagating assertions for the sake of program specialization in a simple functional language called S-Graph. The following assertions as well as their negations are propagated: equalities between variables and constants; equalities between structure components and constants, equalities between variables. A wider set of assertions is propagated in our analysis, and additional assertions are derived. Note that difficulties associated with using the operation meet for propagating assertions are missing from functional languages.

Meyer studied on-line partial evaluation for a Pascal-like language [21]. Andersen developed a partial evaluator for a substantial subset of C [2]. Baier, Glueck, and Zochling created a partial evaluator for Fortran [3]. Blazy and Facon created a Fortran specializer aimed at program understanding [4]. In contrast to these partial evaluation techniques for imperative languages, our specializations are done at the level of control flow graphs.

The structure of the residual code generated by our store-based specialization is similar to that of residual code resulting from the traditional polyvariant program point specialization [6,17,3]. Our residual code is potentially more efficient because of a more precise analysis. The store-based specialization can be combined with various versions of the traditional specialization, which may differ in the way they select variables whose calculations are unfolded. The edge-based specialization generates small residual programs as opposed to the store-based specialization that incorporates loop unrolling [3]. Still, some fragments of these small residual programs are more efficient then traditional residual code. Essentially, both the store-based and edge-based specializations are intermediate cases of driving [16].

Our framework does not fit the mold of existing static analysis formalisms [18,25,10,23]. The closest analysis frameworks to ours are the constant and assertion propagation framework from [24] and the framework utilizing logical expressions whose constituents are equalities about value numbers [15]. The class of formulas tracked in the intra-procedural framework of [24] is substantially narrower. Algorithm BC ignores impossible pairs in-edge/out-edge for conditional branches. Inference capabilities of the analysis algorithm from [24] are much weaker. The framework from [15] does not include interpretation of conditional branches and does not incorporate any inference mechanism. The analysis presented here subsumes conditional constant propagation [26, 7], and number interval propagation with widening [10, 5] (widening is embedded into meet).

Analysis and optimization algorithms for elimination of conditional branches in RTL are proposed in [22]. The analysis algorithm from [22] is applicable to well-structured programs only. The analysis from [22] incorporates 1-step inference which is substantially weaker than algorithm R. The analysis algorithm from [22] is a pessimistic algorithm. Generally, pessimistic algorithms are less powerful than optimistic ones. Yet another advantage of BC over the analysis from [22] is that BC ignores impossible pairs of in/out-edges for conditional branches. The optimization algorithm from [22] may lead to exponential growth of code size.

## References

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

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

[3] R. Baier, R. Glueck, R. Zochling. Partial Evaluation of Numerical Programs in Fortran. *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, 1994, 119-132.

[4] S. Blazy, P. Facon. Partial Evaluation for The Understanding of Fortran Programs. *International Journal of Software Engineering and Knowledge Engineering,* v. 4, 1994, #4, 535-559.

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

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

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

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

[9] C. Consel, S.C. Khoo. Parameterized Partial Evaluation. *ACM Transactions on Programming Languages and Systems*, v. 15, 1993, #3, 463-493.

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

[11] C.N. Fisher, R.J. LeBlanc, Jr. *Crafting A Compiler*, Benjamin/Cummings, 1988.

[12] Y. Futamura, K. Nogi. Generalized partial computation. In: *Partial Evaluation and Mixed Computation* (D. Bjorner, A.P. Ershov, N.D. Jones, Eds.), North-Holland, 1988.

[13] R. Glueck, A. V. Klimov. Occam's Razor in Metacomputation: the Notion of a Perfect Process Tree. *Lecture Notes in Computer Science*, v. 724, 1993, 112-123.

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

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

[16] N. Jones. The Essence of Program Transformation by Partial Evaluation and Driving. *Lecture Notes in Computer Science*, v. 792, 1994, 206-224.

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

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

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

[20] 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.

[21] U. Meyer. Techniques for Partial Evaluation of Imperative Languages. *Proceedings of the ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, 1991, 94-105.

[22] 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.

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

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

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

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