SEMANTICS OF PROGRAM REPRESENTATION GRAPHS

by

G. Ramalingam and Thomas Reps

Computer Sciences Technical Report #900

December 1989

SEMANTICS OF PROGRAM REPRESENTATION GRAPHS

G. RAMALINGAM and THOMAS REPS
University of Wisconsin – Madison

Program representation graphs are a recently introduced intermediate representation form for programs. In this paper, we develop a mathematical semantics for these graphs by interpreting them as data-flow graphs. We also study the relation between this semantics and the standard operational semantics of programs. We show that the semantics of the program representation graphs is more defined than the program semantics and that for states on which a program terminates normally, the PRG semantics is identical to the program semantics.

CR Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors - *compilers, interpreters, optimization*; E.1 [**Data Structures**] *graphs*

General Terms: Theory

Additional Key Words and Phrases: control dependence, flow dependence, program dependence graph, program representation graph, semantics

1. INTRODUCTION

In this paper, we develop a mathematical semantics for program representation graphs (PRGs) and study its relation to the standard (operational) semantics of programs. Program representation graphs are an intermediate representation of programs, introduced by Yang et al. [Yang89a] in an algorithm for detecting program components that exhibit identical execution behaviours. They combine features of static-single-assignment forms (SSA forms) [Shapiro70a, Alpern88a, Cytron89a, Rosen88a] and program dependence graphs (PDGs) [Kuck81a, Ferrante87a, Horwitz89a]. They have also been used in a new algorithm for merging program variants[Yang89b].

Program dependence graphs have been used as an intermediate program representation in various applications such as vectorization, parallelization [Kuck81a], and merging program variants [Horwitz89a]. Horwitz et al. [Horwitz88a] were the first to address the question of whether PDGs were "adequate" as program representations. They showed (for a simplified programming language) that if the program dependence graphs of two programs are isomorphic, the programs are equivalent in the following sense: for any initial state σ, either both programs diverge or both halt with the same final state.

Such an equivalence theorem makes it reasonable to try to develop a semantics for program dependence graphs that is consistent with the program semantics. In contrast to the indirect proof of the equivalence

This work was supported in part by a David and Lucile Packard Fellowship for Science and Engineering, by the National Science Foundation under grant DCR-8552602, by the Defense Advanced Research Projects Agency, monitored by the Office of Naval Research under contract N00014-88-K-0590, as well as by grants from IBM, DEC, and Xerox.

Authors' address: Computer Sciences Department, University of Wisconsin – Madison, 1210 W. Dayton St., Madison, WI 53706.

theorem given in [Horwitz88a], such a semantics would provide a direct proof of the theorem.

Two different semantics have so far been developed for PDGs (and thus each provides a direct proof of the equivalence theorem). Selke [Selke89a] provides a graph rewriting semantics for PDGs. This semantics represents computation steps as graph transformations. The dependence edges are used to make sure that statements are executed in the right order. The store is embedded in the graph. When assignment statements are executed, the relevant portions of the graph are updated to reflect the new value of the corresponding variable. Evaluation of *if* predicates results in deletion of the part of the graph representing the *true* or *false* branch, as appropriate. Evaluation of *while* predicates results in the deletion of the body of the loop or creating a copy of it, as necessary.

Cartwright et al. [Cartwright89a] start with a non-strict generalization of the denotational semantics of the programming language and use a staging analysis to decompose the meaning function into two functions: a *compiler* function that transforms programs into *code trees*, which resemble PDGs, and an *interpreter* function for *code trees*. The interpreter function provides an operational semantics for code trees.

A different (and perhaps more natural) way to develop a semantics for program dependence graphs would be to treat them as graphs of some data-flow programming language and use the conventional operational semantics of such programming languages. Although analogies between PDGs and data-flow graphs have been made previously, this idea has not actually been formalized (*i.e.*, to date no semantics has been developed that interprets PDGs as data-flow graphs). In fact, there are some problems in doing so, as will be explained in Section 4.

In this paper, we show that, with minor modifications, PRGs — as opposed to PDGs — are very naturally interpreted as data-flow graphs. That is, we show how to develop a mathematical semantics for PRGs by formalizing the analogy between PRGs and data-flow graphs. We create a set of possibly mutually recursive equations that, as a function of the initial store, associate a sequence of values with each vertex in the PRG. The semantics of the PRG is defined to be the least fixed point solution of these equations. (This approach is similar to the one taken by Kahn [Kahn74a] in developing a semantics for a parallel programming language.)

The data-flow semantics for PRGs can be restricted so as to give a semantics for PRGs as store-to-store transformers. However, for some applications of PRGs, such as merging program variants, the more general semantic definition is preferable. The more general semantic definition also leads to a stronger form of the equivalence theorem for PRGs that relates the sequences of values computed at corresponding vertices of programs that have isomorphic PRGs.

In particular, we show that (1) the sequence of values computed at any program point (according to the operational semantics) is, in general, a prefix of the sequence associated with that program point by the PRG semantics and (2) for normally terminating program executions the two sequences are identical. This yields the following equivalence theorem: If the PRGs of two programs are isomorphic, then for any initial state σ, either (1) both programs terminate normally, and the sequence of values computed at corresponding vertices are equal or (2) neither program terminates normally and for any pair of corresponding vertices, the sequence of values computed at one of them will be a prefix of the sequence of values computed at the other. Indirect proofs of such equivalence theorems have been previously derived for PDGs [Reps89a] and PRGs [Yang89c]; this paper provides the first direct proof of the result.

The remainder of this paper is organized as follows: Section 2 describes the programming language under consideration. Section 3 defines program representation graphs and Section 4 extends this definition. Section 5 presents the semantics of PRGs, while Section 6 deals with various properties of the standard semantics. Section 7 considers the relation between a program's semantics and its PRG's semantics.

2. The Programming Language Under Consideration

We are concerned with a programming language with the following characteristics: expressions contain only scalar variables and constants; statements are either assignment statements, conditional statements, while-loops, or *end* statements. An *end* statement, which can only appear at the end of a program, names zero or more of the variables used in the program. The variables named in the *end* statement are those whose final values are of interest to the programmer. An example program is shown in the upper left-hand corner of Figure 1.

Our discussion of the language's semantics is in terms of the following informal model of execution. We assume a standard operational semantics for sequential execution; the statements and predicates of a program are executed in the order specified by the program's control flow graph; at any moment there is a single locus of control; the execution of each assignment statement or predicate passes control to a single successor; the execution of each assignment statement changes a global execution state. An execution of the program on an initial state yields a (possibly infinite) sequence of values for each predicate and assignment statement in the program; the i^{th} element in the sequence for program component c consists of the value computed when c is executed for the i^{th} time.

3. Program Representation Graphs

As mentioned previously, PRGs combine features of SSA forms and PDGs. In the SSA form of a program, special assignment statements (ϕ assignments) are inserted so that exactly one assignment to a variable x, either an assignment from the original program or a ϕ assignment, can reach a use of x from the original program. The ϕ statements assign the value of a variable to itself; at most two assignments to a variable x can reach the use of x in a ϕ statement. For instance, consider the following example program fragments:

L_1	$x := 1$		L_1	$x := 1$
	if p **then**			**if** p **then**
L_2	$x := 2$		L_2	$x := 2$
	fi			**fi**
L_4	$y := x + 3$		L_3	$x := \phi_{if}(x)$
			L_4	$y := x + 3$

In the source program (on the left), both assignments to x at L_1 and L_2 can reach the use of x at L_4; after the insertion of "$x := \phi_{if}(x)$" at L_3 (on the right), only the ϕ assignment to x can reach the use of x at L_4. Both assignments to x at L_1 and L_2 can reach the use of x at L_3.

Different definitions of program dependence graphs have been given, depending on the intended application; nevertheless, they are all variations on a theme introduced in [Kuck72a], and share the common feature of having an explicit representation of data dependences. The program dependence graph defined in [Ferrante87a] introduced the additional feature of an explicit representation for control dependences. The program representation graph, defined below, has edges that represent control dependences and one kind of data dependence, called flow dependence.

The program representation graph of a program P, denoted by R_P, is constructed in two steps. First an augmented control flow graph is built and then the program representation graph is constructed from the augmented control flow graph. An example program, its augmented control flow graph, and its program representation graph are shown in Figure 1.

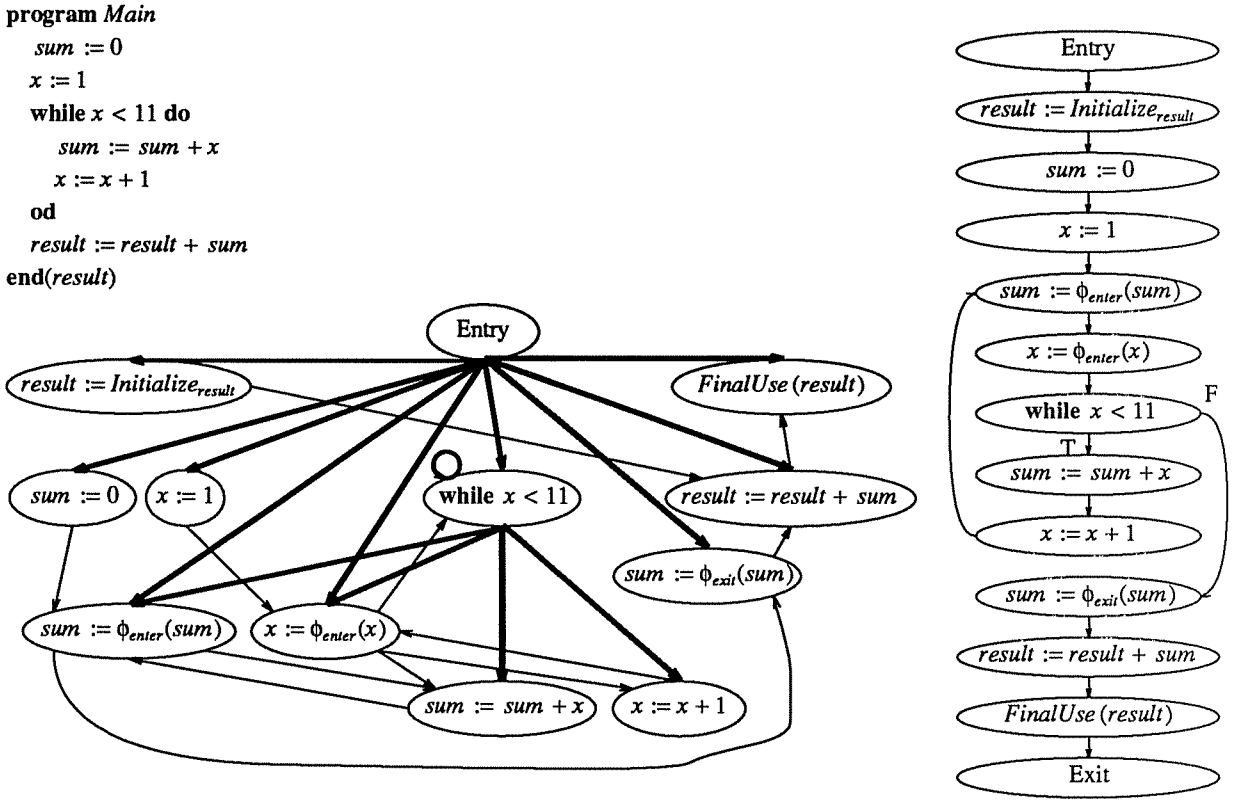

Figure 1. An example program is shown on the top left. This example sums the integers 1 to 10 and adds the sum to the variable *result*. On the right is the augmented control flow graph for the program. Note the absence of *Initialize* and *FinalUse* vertices for *sum* and x and of a ϕ_{exit} vertex for x. On the bottom left is the program representation graph for the program. Note that there is a control dependence edge from the *while* predicate $x < 11$ to itself. The boldface arrows represent control dependence edges; thin arrows represent flow dependence edges. The label on each control dependence edge – *true* or *false* – has been omitted.

Step 1:

The control flow graph[1] of program P is augmented by adding *Initialize*, *FinalUse*, ϕ_{if}, ϕ_{enter}, and ϕ_{exit} vertices, as follows:

(1) A vertex labeled "$x := Initialize_x$" is added at the beginning of the control flow graph for each variable x that may be used before being defined in the program. If there are many *Initialize* vertices for

[1]In control flow graphs, vertices represent the program's assignment statements and predicates; in addition, there are two additional vertices, *Entry* and *Exit*, which represent the beginning and the end of the program. The *Entry* vertex is interpreted as an *if* predicate that evaluates to *true*, and the whole program is interpreted as the *true* branch of the *if* statement. See [Ferrante87a]. For the sake of simplicity, we leave out the edge from *Entry* to *Exit* in the control flow graph.

a program, their relative order is not important as long as they come immediately after the *Entry* vertex.

(2) A vertex labeled "*FinalUse* (x)" is added at the end of the control flow graph for each variable x that appears in the *end* statement of the program. If there are many *FinalUse* vertices for a program, their relative order is not important as long as they come immediately before the *Exit* vertex.

(3) For every variable x that is defined within an *if* statement, and that may be used before being redefined after the *if* statement, a vertex labeled "$x := \phi_{if}(x)$" is added immediately after the *if* statement. If there are many ϕ_{if} vertices for an *if* statement, their relative order is not important as long as they come immediately after the *if* statement.

(4) For every variable x that is defined inside a loop, and that may be used before being redefined inside the loop or may be used before being redefined after the loop, a vertex labeled "$x := \phi_{enter}(x)$" is added immediately before the predicate of the loop. If there are many ϕ_{enter} vertices for a loop, their relative order is not important as long as they come immediately before the loop predicate. After the insertion of ϕ_{enter} vertices, the first ϕ_{enter} vertex of a loop becomes the entry point of the loop.

(5) For every variable x that is defined inside a loop, and that may be used before being redefined after the loop, a vertex labeled "$x := \phi_{exit}(x)$" is added immediately after the loop. If there are many ϕ_{exit} vertices for a loop, their relative order is not important as long as they come immediately after the loop.

Note that ϕ_{enter} vertices are placed inside of loops, but ϕ_{exit} vertices are placed outside of loops.

Step 2:

Next, the program representation graph is constructed from the augmented control flow graph. The vertices of the program representation graph are those in the augmented control flow graph (except the *Exit* vertex). Edges are of two kinds: control dependence edges and flow dependence edges.

A control dependence edge from a vertex u to a vertex v, denoted by $u \longrightarrow_c v$, means that, during execution, whenever the predicate represented by u is evaluated and its value matches the label – *true* or *false* – on the edge to v, then the program component represented by v will eventually be executed if the program terminates normally. The source of a control dependence edge is the *Entry* vertex or a predicate vertex.

- There is a control dependence edge from *Entry* to a vertex v if v occurs on every path from *Entry* to *Exit* in the augmented control flow graph. This control dependence edge is labeled *true*.
- There is a control dependence edge from a predicate vertex u to a vertex v if, in the augmented control flow graph, v occurs on every path from u to *Exit* along one branch out of u but not the other. This control dependence edge is labeled by the truth value of the branch in which v always occurs.

Note that there is a control dependence edge from a *while* predicate to itself. Methods for determining control dependence edges for programs with unrestricted flow of control are given in [Ferrante87a, Cytron89a]; however, for our restricted language, control dependence edges can be determined in a simpler fashion: Except for the extra control dependence edge incident on a ϕ_{enter} vertex, the control dependence edges merely reflect the nesting structure of the program.

A flow dependence edge from a vertex u to a vertex v, denoted by $u \longrightarrow_f v$, means that the value produced at u may be used at v. There is a flow dependence edge $u \longrightarrow_f v$ if there is a variable x that is

assigned a value at u and used at v, and there is an x-definition-free path from u to v in the augmented control flow graph. The flow dependence edges of a program representation graph can be computed using data-flow analysis.

The *imported variables* of a program P, denoted by Imp_P, are the variables that might be used before being defined in P, *i.e.*, the variables for which there are *Initialize* vertices in the *PRG* of P.

Textually different programs may have isomorphic program representation graphs. However, it has been shown that if two programs have isomorphic program representation graphs, then the programs are semantically equivalent [Yang89c]:

THEOREM. (EQUIVALENCE THEOREM FOR PROGRAM REPRESENTATION GRAPHS). *Suppose P and Q are programs for which R_P is isomorphic to R_Q. If σ is a state on which P halts, then for any state σ' that agrees with σ on the imported variables of P, (1) Q halts on σ', (2) P and Q compute the same sequence of values at each corresponding program component, and (3) the final states of P and Q agree on all variables for which there are final-use vertices in R_P and R_Q.*

4. EXTENSIONS TO PROGRAM REPRESENTATION GRAPHS

Our aim is to treat PRGs as pure data-flow graphs. Data-flow graphs are a model of parallel computation, where vertices represent computing agents and edges represent unidirectional communication channels. Values computed at one vertex are transmitted to other vertices along the edges. In this model, the sequence of values "flowing" along an edge out of vertex u is a function of the sequences of values "flowing" along edges incident on vertex u.

The trouble with treating PDGs (as opposed to PRGs) as data-flow graphs is that multiple definitions of a variable may reach a vertex. In contrast, vertices in data-flow graphs tend to have only one incident edge per variable, the only exception being certain control vertices that choose the value in one of two incident edges based on a boolean input. In contrast with PDGs, PRGs resemble data-flow graphs in this respect — normally only one definition of any variable reaches any vertex. The exceptions are the ϕ_{if} and ϕ_{enter} vertices which are reached by two definitions of a variable. Both ϕ_{if} and ϕ_{enter} vertices are associated with predicate nodes, and are similar to the control vertices in data-flow graphs.

There is a small problem in treating PRGs as data-flow graphs. If $u \longrightarrow_f v$ is a data dependence, then a particular value computed at u may be used zero or more times at v! However, in data-flow graphs a value flowing along an edge is consumed exactly once. In order to get around this problem, we introduce several new kinds of ϕ nodes that can consume unused data or duplicate them a certain number of times. These extra nodes make it possible to view PRGs as data-flow graphs and simplify the definition of PRG semantics.

The essential idea is to replace all data dependences $u \longrightarrow_f v$ that can cause the above-mentioned problem by two data dependences $u \longrightarrow_f w$ and $w \longrightarrow_f v$, where w is an appropriate ϕ node, as described below.

(1) Let t be an *if* statement predicate and u a vertex outside the *if* statement. If there exists at least one vertex v such that (1) $u \longrightarrow_f v$ is an edge in the PRG and (2) v is either in the *true* branch of the *if* statement or is a ϕ_{if} vertex associated with the *if* statement such that the definition u reaches v around the *true* branch, then introduce a ϕ_T vertex for the variable defined in u. Let w denote the new vertex. Add the control dependence edge $t \longrightarrow_c w$ labelled *true* and the data dependence edge $u \longrightarrow_f w$.

For each vertex v satisfying the above condition, replace $u \rightarrow_f v$ by the edge $w \rightarrow_f v$. (Note that if $u \rightarrow_f v$ is a data dependence of the above form, then the value computed at u gets used at v only if the *if* predicate evaluates to *true* at the appropriate instance. In the data-flow semantics, the ϕ_T vertices act as filters that transmit only those values that correspond to the *if* predicate evaluating to *true*.) See Figure 2 for an illustration of this definition. Similarly, ϕ_F vertices are introduced in the *false* branches of *if* statements.

(2) Let t be a *while* statement predicate and u a vertex outside the *while* statement. If there exists at least one non-ϕ_{enter} vertex v inside the *while* statement such that $u \rightarrow_f v$ is an edge in the PRG, then introduce a ϕ_{copy} vertex for the variable defined in u. Let w denote the new vertex. Add control

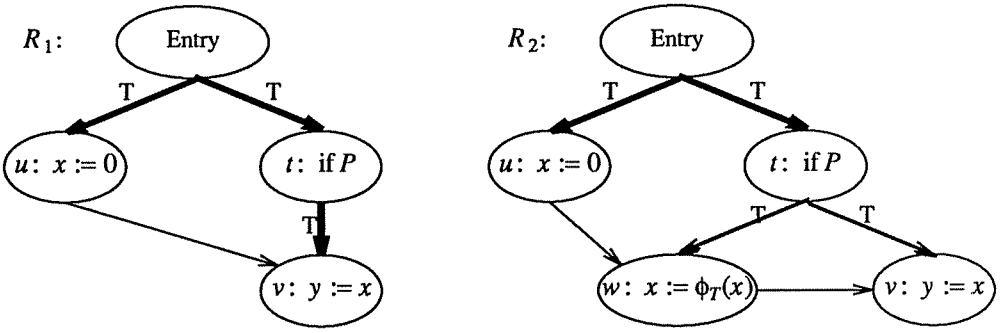

Figure 2. The above example shows how ϕ_T vertices are introduced into PRGs.

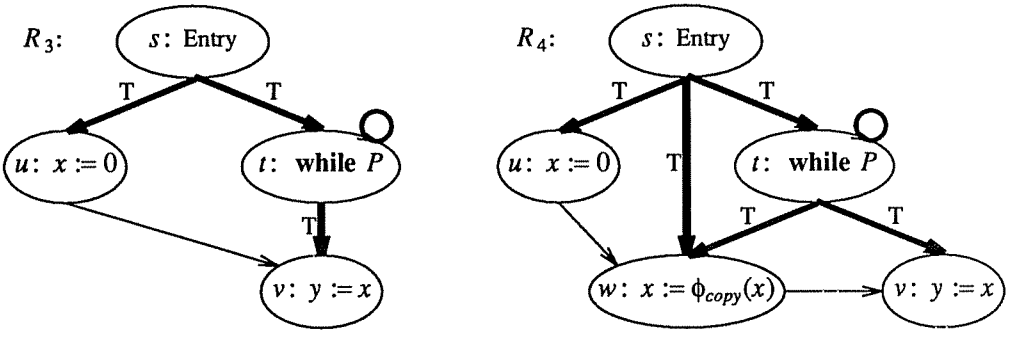

Figure 3. The above example shows how ϕ_{copy} vertices are introduced into PRGs. Note that a ϕ_{while} vertex will subsequently be introduced between w and v.

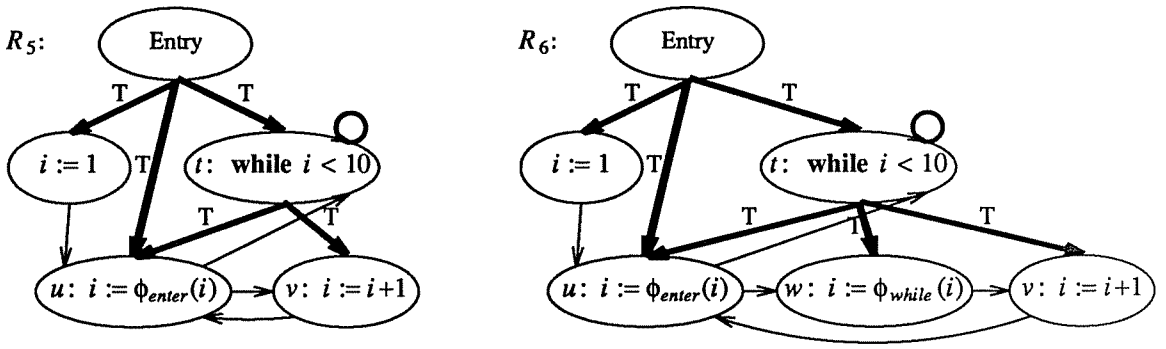

Figure 4. The above example shows how ϕ_{while} vertices are introduced into PRGs.

dependences $t \rightarrow_c w$ and $s \rightarrow_c w$, where s is t's parent, just as for a ϕ_{enter} vertex. Add data dependence $u \rightarrow_f w$. For each vertex v satisfying the above condition, replace $u \rightarrow_f v$ by $w \rightarrow_f v$. (Note that in such cases a value computed at u may be used several times at v. The ϕ_{copy} vertex creates multiple copies of the value computed at u, one for each evaluation of the *while* predicate during the execution of the loop.) See Figure 3 for an illustration of this definition.

(3) Let t be a *while* statement predicate and u a ϕ_{enter} or ϕ_{copy} vertex associated with the *while* statement. If there exists at least one vertex $v \neq t$ inside the *while* statement such that $u \rightarrow_f v$ is an edge in the PRG, then introduce a ϕ_{while} vertex w corresponding to the relevant variable. Add the control dependence $t \rightarrow_c w$ labelled *true* and the data dependence $u \rightarrow_f w$. For each vertex v satisfying the above condition, replace $u \rightarrow_f v$ by $w \rightarrow_f v$. (Note that in such cases the value computed at u gets used at v only if the *while* predicate evaluates to *true* at the corresponding instance. The ϕ_{while} vertex filters out the values corresponding to an evaluation of the *while* predicate to *false*.) See Figure 4 for an illustration of this definition.

The above transformations are performed in the following order: Traverse the control-dependence subtree of the PRG in a top-down fashion. For each predicate vertex t, for each suitable vertex u, perform either (1) or (2) and (3) (in that order) as appropriate.

Let us call the resulting structure an extended PRG. Note that the process guarantees that if there is any data dependence $u \rightarrow_f v$ and v is a non-ϕ vertex, then u and v have the same set of control dependence predecessors, *i.e.*, u and v execute under the same conditions. Thus, barring nontermination or abnormal termination, each value computed at u gets used exactly once at v. (Normally, a control dependence predecessor of a vertex u is just a vertex v such that there exists a control dependence edge $v \rightarrow_c u$. But occasionally, as above, we use the phrase to refer to a pair $<v,b>$ such that there exists a control dependence edge $v \rightarrow_c u$ labelled b).

The above extensions may be viewed as describing a graph-transformation function **E** - thus, if G is a PRG, then **E**(G) is the extended PRG. In the following section, we present a semantics for extended PRGs, represented by the semantic function **M**. The semantics of the (unextended) PRG G is then defined to be **M**(**E**(G)).

Let G be the PRG of a program P. We would like to relate the PRG semantics of the G to the standard operational semantics of program P. To do this, we augment program P with ϕ-statements so that there is a one-to-one correspondence between the statements and predicates of the program so obtained (denoted P′) and the vertices of E(G). This is done just as in Section 3, with appropriate ϕ-statements being added to correspond to the ϕ vertices introduced in this section. (Thus, what we get is really an augmented control flow graph, which has a standard operational semantics.) This simplifies various proofs relating the PRG semantics to the program semantics. Since each ϕ statement is an assignment of some variable to itself, the introduction of such statements hardly changes the standard semantics of the program. Consequently, the results we derive relating the semantics of E(G) to the semantics of extended program do relate the PRG semantics to the standard program semantics.

Here, it should be noted that E(G) may not be the PRG of the program P′. More precisely, the data dependence edges in E(G) may not correspond to the true data dependences in P′. For instance, consider the PRG G shown in Figure 5. G is the PRG of both programs P_1 and P_2, shown in augmented form below. The extended PRG E(G) turns out to be the PRG of extended program P_2' but not of P_1'.

L_1	$x := 0$	L_1	$x := 0$
	if p **then**	L_4	$z := x$
L_2	$x := \phi_T(x)$		**if** p **then**
L_3	$y := x$	L_2	$x := \phi_T(x)$
	fi	L_3	$y := x$
L_4	$z := x$		**fi**
	Program P_1'		Program P_2'

However, this difference between the actual dependences and the edges in the extended PRG causes no problem, as shown later.

From now on the terms PRGs and programs will refer to extended PRGs (like E(G)) and extended programs (like P′) respectively.

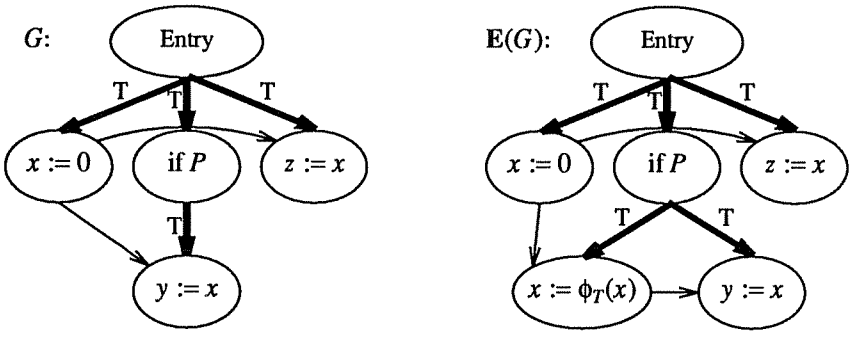

Figure 5. An example of how extended PRGs may not represent the true data dependences of the extended program.

5. PRG SEMANTICS

5.1. Notation

Let u be a vertex in an extended PRG G. The vertex type of u must be one of { *assignment*, *if*, *while*, ϕ_T, ϕ_F, ϕ_{if}, ϕ_{while}, ϕ_{enter}, ϕ_{exit}, ϕ_{copy}, *Entry*, *Initialize*, *FinalUse*} and will be denoted by *typeOf*(u). If u is an *assignment*, *if* or *while* vertex, then *functionOf*(u), a function of n variables, will represent the n-variable expression in vertex u. When the variables in the expression are abstracted to create a function, they have to be done so in some particular order. The variables, in the same order, are denoted by $var_1(u)$, ..., $var_n(u)$. The data dependence predecessor corresponding to $var_i(u)$ at u (where u is an *assignment*, *if* or *while* vertex) is denoted by $dataPred_i(u)$. If u has a unique data dependence predecessor, *dataPred*(u) will denote the predecessor vertex. Let *parent*(u) denote the unique control dependence predecessor of vertex u, ignoring self loops; in the case that u is a ϕ_{enter} or a ϕ_{copy} vertex, *parent*(u) denotes the corresponding *while* predicate vertex. If $u \longrightarrow_c v$ is a control dependence edge, *label*(u,v) will denote the label (*true* or *false*) on that control dependence edge. Let *controlLabel*(u) denote *label*(*parent*(u),u). If u is an *Initialize*, *FinalUse* or ϕ vertex, then *varOf*(u) denotes the corresponding variable.

If u is a ϕ_{if}, ϕ_T or ϕ_F vertex, then *ifNode*(u) will denote the corresponding *if* predicate vertex. Similarly, if u is some ϕ node associated with a *while* loop, *whileNode*(u) will denote the corresponding *while* predicate vertex. If u is a ϕ_{enter} vertex, then *innerDef*(u) and *outerDef*(u) denote the definitions that reach u from inside and outside the loop, respectively. If u is a ϕ_{if} vertex, then *trueDef*(u) and *falseDef*(u) are defined similarly. Let CDP(u) denote the set of control dependence predecessorsof u, along with their associated labels.

Let *Var* denote the domain of variable names. Let *Val* denote the basic domain of first-order values manipulated by the programs - it includes the domain of *booleans* and any other domains of interest.

$$Val = boolean + integer + real + ...$$

The domain *Sequence*, consisting of finite and infinite sequences of values belonging to *Val*, is defined by the following recursive domain equation, where *NIL* denotes the domain consisting of the single value *nil*.

$$Sequence = (NIL + (Val \times Sequence))_\perp$$

The *cons* operator is denoted by $\cdot$, as in $head \cdot tail$. Elements of the *Sequence* domain may be classified into three kinds: Finite sequences (like $a_1 \cdot a_2 \cdot \cdots \cdot a_n \cdot nil$), infinite sequences ($a_1 \cdot a_2 \cdot \cdots$) and sequences whose suffix is unknown or undefined (like $a_1 \cdot a_2 \cdot \cdots \cdot a_n \cdot \perp$). (In particular, errors like "division-by-zero" lead to $\perp$-terminated sequences.) If a sequence X has at least i elements, then X(i) is defined to be the i^{th} element of X. Otherwise, it is undefined. X($i..j$), similarly, denotes the corresponding subsequence of X, if all those elements are defined. Otherwise, it is undefined. If X contains at least j occurrences of the value x, then *index*(X,j,x) is defined and denotes the position in sequence X of the j^{th} occurrence of x. #(X,j,x) denotes the number of occurrences of x in X(1..j). A sequence X is said to be a prefix of sequence Y iff for every X(i) that is defined, Y(i) is defined and equal to X(i).

The meaning function **M** we want to define belongs to the domain $PRG \longrightarrow Store \longrightarrow Vertex \longrightarrow Sequence$.

5.2. The Semantics

Let G be the PRG under consideration and σ the initial store. For each vertex u in G, we define a sequence S(u) by an equation, which depends on the type of the vertex u. This set of equations can be combined into one recursive equation of the form

$$s = \mathrm{F}\, s$$

where s combines all the sequences and hence effectively belongs to the domain *Vertex* $\longrightarrow$ *Sequence*. The least fixed point S of this equation is given by

$$S = \bigsqcup_{i=0}^{\infty} F^i(\perp).$$

This least fixed point is taken to be the semantics of the given PRG with respect to the given store, i.e. $\mathbf{M}[G]\sigma = S$.

The equations are described below.

If u is the *Entry* vertex,

$$S(u) = \text{true} \cdot nil$$

If u is an *Initialize* vertex,

$$S(u) = \sigma(varOf(u)) \cdot nil$$

If u is a *FinalUse* vertex,

$$S(u) = S(dataPred(u))$$

If u is a ϕ_T vertex,

$$S(u) = \text{select}(\text{ true}, S(parent(u)), S(dataPred(u)))$$

where

$$\text{select}(x, y \cdot tail_1, z \cdot tail_2) = \textbf{if } (x = y) \textbf{ then } z \cdot \text{select}(x, tail_1, tail_2) \textbf{ else } \text{select}(x, tail_1, tail_2)$$

$$\text{select}(x, nil, z) = nil$$

$$\text{select}(x, y, nil) = nil$$

Note: The value of select(x,s_1,s_2) is the sequence consisting of the values $s_2(i)$ such that $s_1(i)$ is the value x. More formally, let s_3 = select(x,s_1,s_2). If $s_1(1..j)$ and $s_2(1..j)$ are defined and $j = index(s_1,i,x)$, then $s_3(i)$ is defined and equal to $s_2(j)$. Conversely, if $s_3(i)$ is defined, then there must be a j such that $s_1(1..j)$ and $s_2(1..j)$ are defined and $j = index(s_1,i,x)$.

If u is a ϕ_F vertex,

$$S(u) = \text{select}(\text{ false}, S(parent(u)), S(dataPred(u)))$$

If u is a ϕ_{if} vertex,

$$S(u) = \text{merge}(\text{ } S(ifNode(u)), S(trueDef(u)), S(falseDef(u)) \text{)}$$

where

$$\text{merge}(\text{ true} \cdot tail_1, x \cdot tail_2, s) = x \cdot \text{merge}(tail_1, tail_2, s)$$

$$\text{merge}(\text{ true} \cdot tail_1, nil, s) = nil$$

$$\text{merge}(\text{ false} \cdot tail_1, s, x \cdot tail_2) = x \cdot \text{merge}(tail_1, s, tail_2)$$

$$\text{merge}(\text{ false} \cdot tail_1, s, nil) = nil$$

$$\text{merge}(\text{ } nil, y, z) = nil$$

Note: Let s_4 = merge(s_1,s_2,s_3). s_4 is the sequence obtained by merging the two sequences s_2 and s_3 according to s_1, a sequence of boolean values. More formally, $s_4(i)$ is defined iff $s_1(1..i)$ is defined and $s_2(1..j)$ and $s_3(1..i\text{-}j)$ are defined, where $j = \#(s_1,i,true)$ and $i\text{-}j = \#(s_1,i,false)$. Further, if the

latter conditions are true, $s_4(i)$ will equal $s_2(j)$ if $s_1(i)$ equals *true* and $s_3(i\text{-}j)$ if $s_1(i)$ equals *false*.

If u is a ϕ_{exit} vertex,

$S(u)$ = select(false, S(*whileNode*(u)), S(*dataPred*(u)))

If u is a ϕ_{while} vertex,

$S(u)$ = select(true, S(*whileNode*(u)), S(*dataPred*(u)))

If u is a ϕ_{enter} vertex,

$S(u)$ = whileMerge(S(*whileNode*(u)), S(*innerDef*(u)), S(*outerDef*(u)))

where

whileMerge(s_1, s_2, $x \cdot$ tail) = $x \cdot$ merge(s_1,s_2,tail)

whileMerge(s_1, s_2, *nil*) = *nil*

Note: From the definition, it can be seen that the function whileMerge is quite related to merge. Let s = whileMerge(s_1,s_2,s_3). $s(1)$ is defined and equal to $s_3(1)$ iff $s_3(1)$ is defined. For $i \geq 1$, $s(i+1)$ is defined iff $s_1(1..i)$ is defined and $s_2(1..j)$ and $s_3(1..i+1\text{-}j)$ are defined, where j = #(s_1,i,*true*) and $i\text{-}j$ = #(s_1,i,*false*). If the latter conditions are true, then $s(i+1)$ will equal $s_2(j)$ if $s_1(i)$ equals *true* and $s_3(i+1\text{-}j)$ if $s_1(i)$ equals *false*.

If u is a ϕ_{copy} vertex,

$S(u)$ = whileCopy(S(*whileNode*(u)), S(*dataPred*(u)))

where

whileCopy(s, $x \cdot$ tail) = $x \cdot$ copy(s, $x \cdot$ tail)

whileCopy(s, *nil*) = *nil*

copy(true $\cdot$ $tail_1$, $x \cdot tail_2$) = $x \cdot$ copy($tail_1$, $x \cdot tail_2$)

copy(false $\cdot$ $tail_1$, $x \cdot tail_2$) = whileCopy($tail_1$, $tail_2$)

Note: whileCopy(s_1,s_2) is the sequence obtained by duplicating each element $s_2(i)$ (of the sequence s_2) n+1 times, where n is the number of occurrences of 'true' between the $i-1^{th}$ and i^{th} occurrence of 'false' in the sequence s_1. More formally, if s = whileCopy(s_1,s_2), then (1) $s(1)$ is defined and equal to $s_2(1)$ iff $s_2(1)$ is defined and (2) $s(i+1)$ is defined iff $s_1(1..i)$ is defined and $s_2(1..j)$ is defined, where j = #(s_1,i,*false*)+1. If the latter conditions hold, then $s(i+1)$ equals $s_2(j)$.

The remaining possibilities are that u is an *assignment* vertex, an *if* predicate vertex, or a *while* predicate vertex. In these cases, if u has n data dependence predecessors, where $n > 0$, then

$S(u)$ = map (*functionOf*(u)) (S($dataPred_1(u)$), ... S($dataPred_n(u)$))

where

map (f)($x_1 \cdot tail_1$, $x_2 \cdot tail_2$, ..., $x_n \cdot tail_n$) = $f(x_1, x_2, ..., x_n)$. map (f) ($tail_1$, $tail_2$, ..., $tail_n$)

map (f)(*nil*, *nil*, ..., *nil*) = *nil*

Note: Let f be an n variable function. Let s = map (f)(s_1, ..., s_n). Then, $s(i)$ is defined and equal to $f(s_1(i), ..., s_n(i))$ iff $s_1(1..i)$, ..., $s_n(1..i)$ are all defined.

If n is 0, then the expression in vertex u is a constant-valued expression. This case divides into two subcases. If u is anything other than a true-valued *while* predicate vertex, then

$S(u)$ = replace (*controlLabel*(u), *functionOf*(u), S(*parent*(u)))

where

replace(x, y, z $\cdot$ tail) = **if** ($x = z$)

then y · replace(x,y,tail)
else replace(x,y,tail)

replace(x, y, *nil*) = *nil*

Note: Let s_2 = replace(x,y,s_1). s_2 is essentially a sequence consisting of as many 'y's as s_1 has 'x's. Thus, all elements in s_1 which do not equal x are ignored, and the remaining elements are each replaced by y. More formally, $s_2(i)$ is defined and equal to y iff *index*(s_1,i,x) is defined.

If u is a true-valued *while* predicate vertex, then

S(u) = whileReplace(*controlLabel*(u), S(*parent*(u)))

where

whileReplace(x, *nil*) = *nil*

whileReplace(x, z · tail) = **if** (x = z)
then infiniteTrues
else whileReplace(x,tail)

infiniteTrues = true · infiniteTrues

Note: whileReplace(x, s) is an infinite sequence of 'true's if the value x occurs in the sequence s, and the empty sequence otherwise.

6. STANDARD SEMANTICS

Consider the sequential execution of a program P on initial store σ, under the standard operational semantics. Let I denote the sequence of program points executed - *i.e.*, I(i) is the program point executed in the i^{th} step. Let V denote the corresponding sequence of values computed. Thus, V(i) denotes the value computed during the i^{th} execution step. If a program point u executes at least i times, then *step*(u,i) denotes the step number at which u executes for an i^{th} time - *i.e.*, *step*(u,i) = *index*(I,i,u).

Let A(u) denote the (possibly infinite) sequence of values computed at program point u. Thus, A(u)(i) is defined iff *step* (u,i) is defined, in which case it equals V(*step*(u,i)). Let *value*(x,u,i) denote the value of the variable x at the begining of the *step* (u,i)th execution step. We will be interested in *value*(x,u,i) only if x is used at program point u. We now observe some of the properties that hold among the various sequences. Our aim is to express A(u)(i) in terms of values computed before *step*(u,i).

All ϕ statements, *Initialize* statements and *FinalUse* statements represent an assignment of a variable to itself. Other statements u compute the value *functionOf*(u)(*var*$_1$(u), ..., *var*$_n$(u)). This gives us the following property.

Property 1.

A(u)(i) = value(varOf(u),u,i) if u is a ϕ statement, *Initialize* statement or *FinalUse* statement,
= functionOf(u)(value(*var*$_1$(u),u,i), ..., value(*var*$_n$(u),u,i)) otherwise.

In the standard semantics, the store is used to communicate values between statements in the program. The following property follows directly from the way a store is used.

Property 2.

Let A = { j | 1 ≤ j < *step*(u,i) and I(j) assigns to variable x } and
let max(A) denote the maximum element in the set A. Then,
value(x,u,i) = V(max(A)) if A is non-empty
= σ(x) otherwise, where

The introduction of *Initialize* statements guarantees the following property.

Property 3. If x is some variable used at u and $step(u,i)$ is defined, then the set $\{ j \mid 1 \le j < step(u,i)$ and $I(j)$ assigns to variable $x \}$ is empty iff u is an Initialize statement.

Note that since the programming language has no aliasing mechanism, we can talk about assignments to variables rather than locations. It also makes it possible to compute statically the variable to which a statement dynamically assigns a value. Let $RD(u,x)$ denote the set of reaching definitions of variable x at statement u. The following is a property of reaching definitions.

Property 4. If u is not an Initialize statement and x is some variable used at u, then $\max(\{ j \mid 1 \le j < step(u,i)$ and $I(j)$ assigns to variable $x\})$ is equal to $\max(\{ j \mid 1 \le j < step(u,i)$ and $I(j) \in RD(u,x) \})$.

The preceding three properties imply the following.

Property 5. If x is some variable used at u, then

$value(x,u,i) = \sigma(x)$ if u is an Initialize statement,

$= V(\max \{ j \mid 1 \le j < step(u,i)$ and $I(j) \in RD(u,x) \})$ otherwise.

Let $DDP_G(u,x)$ denote the set of data dependence predecessors corresponding to variable x of vertex u in graph G. We drop the subscript G if G is the extended PRG. If G is the PDG or PRG of the extended program, then $DDP_G(u,x) = RD(u,x)$, by definition (assuming that x is used at program point u). However, this need not be true if G is the extended PRG, as observed earlier. Yet, the data dependence edges in the extended PRG are a sufficient enough approximation to the reaching definitions for the following property to hold.

Property 6. If u is not an *Initialize* statement and x is some variable used at u, then

$value(x,u,i) = V(\max \{ j \mid 1 \le j < step(u,i)$ and $I(j) \in DDP(u,x) \})$

The justification for the above claim follows. Let $k = \max \{ j \mid 1 \le j < step(u,i)$ and $I(j) \in DDP(u,x) \}$. Since $I(k) \in DDP(u,x)$, $I(k)$ must assign to x. Now, for any j such that $k < j < step(u,i)$, if $I(j)$ is an assignment to x, then $I(j)$ must be one of the new ϕ-statements introduced in the extension of PRGs (Section 4). (If not, consider the maximum j such that $k < j < step(u,i)$, $I(j)$ is an assignment to x and $I(j)$ is not one of the new ϕ-statements. Then, the data dependence $I(j) \longrightarrow_f u$ must have been in the original PRG. Consequently, either $I(j) \longrightarrow_f u$ must be present in the extended PRG, or there must be an m such that $j < m < step(u,i)$ and $I(m) \longrightarrow_f u$ is present in the extended PRG. Either way, we have a contradiction with the maximality of k in its definition.) Since all the new ϕ-statements are assignments of a variable to itself, the above result follows.

Observe that all statements other than ϕ_{enter} and ϕ_{if} statements have only one data dependence predecessor per variable. In such cases the above equation may be simplified to yield the following property.

Property 7. If $DDP(u,x) = \{v\}$, then

$value(x,u,i) = V(\max \{ j \mid 1 \le j < step(u,i)$ and $I(j) = v \})$

$= V(\max \{ step(v,k) \mid step(v,k) < step(u,i) \})$

$= A(v)(k)$ where k is such that $step(v,k) < step(u,i) < step(v,k+1)$

The notation $step(x,i) < step(y,j)$ essentially means that program point y executes for the j^{th} time only after program point x executes for the i^{th} time. However, we will also say that $step(x,i) < step(y,j)$ even if y does not execute j times.

The following properties concern control dependence and help identify the value of k in property 7. Observe that if v is a data dependence predecessor of u and u and v are non-ϕ statements, then u and v have the same control dependence predecessor and v occurs to the left of u in the program's abstract syntax tree. More generally, if v is any kind of data dependence predecessor of u and u is a non-ϕ vertex, then u and v have the same control dependence predecessors and v dominates u in the extended control-flow graph.

Property 8. If CDP(u) = CDP(v) and v dominates u, then $step(v,i) < step(u,i) < step(v,i+1)$, for all i. Further, if the program execution terminates normally, u and v execute the same number of times.

The previous two properties combined with property 1 gives us the following result.

Property 9. If u is an *assignment* statement, *if* predicate or *while* predicate and executes i times, then

$$A(u)(i) = functionOf(u)(A(dataPred_1(u))(i), \ldots A(dataPred_n(u))(i))$$

Let u have only one control dependence predecessor, say v. Let this control dependence edge be labelled *true*. Then, barring nontermination or abnormal termination, u is executed exactly once each time v is evaluated to *true*. More formally, we can say:

Property 10. Let $v \longrightarrow_c u$, labelled *true*, be the sole control dependence edge incident on u. Then, if $A(u)(i)$ is defined, $j = Index(A(v),i,true)$ must be defined and $step(v,j) < step(u,i)$. Conversely, if $Index(A(v),i,true)$ is defined, then $A(u)(i)$ must be defined, barring nontermination or abnormal termination.

The above property can be extended to state that $step(u,i)$ occurs "soon after" $step(v,j)$ - i.e., before any other statement at the same nesting level as v can be executed. Let w denote some vertex with the same control dependence predecessors as v and occuring to the left of v. (As a specific example, let u be a ϕ_T vertex. Let v be $parent(u)$ and w be $dataPred(u)$.) It is easy to see that $step(w,j) < step(u,i) < step(w,j+1)$. This gives us the following property.

Property 11. Let u be a ϕ_T vertex. Let v denote $parent(u)$ and w denote $dataPred(u)$. If $A(u)(i)$ is defined, then $j = Index(A(v),i,true)$ must be defined and

$$step(w,j) < step(v,j) < step(u,i) < step(w,j+1)$$

and consequently, from properties 1 and 7,

$$A(u)(i) = A(w)(j)$$

A similar extension of property 10 to consider a vertex w with the same control dependence predecessors as v and occurring to the right of v yields the following property.

Property 12. Let v be an *if* predicate and let $v \longrightarrow_c u$, labelled *true*, be the only control dependence edge incident on u. Let w be a ϕ_{if} vertex associated with v. Let $j = \#(A(v),i,true)$. Then $step(u,j) < step(w,i) < step(u,j+1)$.

Related versions of the above two properties may be obtained by replacing *true* by *false* and T by F. The following property of ϕ_{if} vertices is obtained from properties 1, 6 and 12.

Property 13. Let u be a ϕ_{if} vertex. Let v be $ifNode(u)$, x be $trueDef(u)$ and y be $falseDef(u)$. If $A(u)(i)$ is defined, then

$$A(u)(i) = \textbf{if } A(v)(i) \textbf{ then } A(x)(j) \textbf{ else } A(y)(i\text{-}j)$$

where $j = \#(A(v),i,true)$.

A formal derivation of the above property follows. Assume $A(v)(i)$ is *true*. Let $\#(A(v),i,true)$ be j. Obviously, $\#(A(v),i\text{-}1,true) = j\text{-}1$, while $\#(A(v),i,false) = \#(A(v),i\text{-}1,false) = i\text{–}j$. Hence, from property 12,

$$step(y,i\text{-}j) < step(u,i\text{-}1) < step(x,j) < step(u,i) < step(x,j+1)$$

$$step(u,i) < step(y,i\text{-}j+1)$$

Properties 1 and 6 imply that $A(u)(i)$ must be $A(x)(j)$. Similarly, if $A(v)(i)$ is *false*, then $A(u)(i)$ must be $A(y)(i\text{-}j)$.

The following property concerns the execution behaviour of a ϕ_{enter} vertex. Here, it is useful to consider the execution of the whole loop (rather than just the loop predicate). The loop completes an execution when the loop predicate evaluates to *false*. Suppose the loop predicate v has been executed i times. Then, the number of times the loop has completed an execution is given by $\#(A(v),i,false)$.

Property 14. Let u be a ϕ_{enter} vertex. Let v be *whileNode*(u), x and y be outerDef(u) and innerDef(u) respectively. Let w be the parent of x and v. Let the control dependences $w \longrightarrow_c v$ and $w \longrightarrow_c u$ be labelled *true*. If $i > 1$, then the following hold true

$step(x,1) < step(u,1) < step(y,1)$

$step(u,i\text{-}1) < step(v,i\text{-}1) < step(u,i)$

$step(x,j) < step(u,i) < step(x,j\text{+}1)$ where

$j = \#(A(v),i\text{-}1,false)\text{+}1$

$step(y,j) < step(u,i) < step(y,j\text{+}1)$ where

$j = \#(A(v),i\text{-}1,true)$

In particular, from property 6, if $A(u)(1)$ is defined, then

$A(u)(1) = A(x)(1)$

and for $i > 1$, if $A(u)(i)$ is defined, then

$A(u)(i) =$ **if** $A(v)(i\text{-}1)$ **then** $A(y)(i\text{-}j)$ **else** $A(x)(j)$

where $j = \#(A(v),i\text{-}1,false)\text{+}1$.

The derivation of the above property is very similar to the derivation of property 13.

The following property concerns ϕ_{copy} vertices. It is similar to, though simpler than, the previous property.

Property 15. Let u be a ϕ_{copy} vertex. Let v denote *whileNode*(u), and w *dataPred*(u). Let $j = \#(A(v),i\text{-}1,false)\text{+}1$ Then,

$step(u,i\text{-}1) < step(v,i\text{-}1) < step(u,i)$,

$step(w,j) < step(u,i) < step(w,j\text{+}1)$

and if $A(u)(i)$ is defined, it must be equal to $A(w)(j)$.

7. RELATION TO PROGRAM SEMANTICS

Now, we consider the relation between the semantics of the PRG of a program, as defined earlier, and the standard operational semantics of the program. We show that in general the sequence $S(u)$ (which is defined by the PRG semantics) may be more defined than the sequence $A(u)$ (the sequence of values computed by program point u, as defined by the operational semantics of the program) - or more formally, that $A(u)$ will be a prefix of $S(u)$. However, for input stores on which the program terminates normally, the sequence $S(u)$ will be shown to be equal to the sequence $A(u)$.

This difference in the case of nonterminating (or abnormally terminating) program execution maybe explained as follows. Data-flow semantics exposes and exploits the parallelism in programs. The eager or data-driven evaluation semantics lets a program point execute as soon as the data it needs is available. In the standard sequential execution of a program, however, the execution of a program point u may have to be delayed until completion of execution of some other part of the program, even if the result of that computation is unnecessary for the computation to be done at u. And, if that computation never terminates or terminates abnormally, execution of program point u does not occur.

Let $S(u)$ denote the least fixed point solution of the set of recursive equations for the PRG of program P and initial store σ as defined in Section 5. As observed earlier, the set of equations can be combined into one recursive equation of the form

$s = F\,s$

Let $S^k(u)$ denote $F^k(\perp)(u)$, the k^{th} approximation to the solution at vertex u. Now we are ready to state a sequence of lemmas and theorems that relate the standard operational semantics and the PRG semantics.

LEMMA. *Let G be the extended PRG of a program P and σ an input store. Let S(u) denote* $\mathbf{M}[G](\sigma)(u)$, $S^k(u)$ *denote the* k^{th} *approximation to S(u) and A(u) denote the sequence of values computed at program point u for input σ under the standard operational semantics. If A(u)(i) is defined, then there exists a k such that* $S^k(u)(i)$ *is defined and equal to A(u)(i).*

PROOF. Observe that S^k is monotonic in k. Hence the lemma is equivalent to the following stronger claim: if $A(u)(i)$ is defined, then there exists a k such that $S^n(u)(i)$ is defined and equal to $A(u)(i)$, for all $n \geq k$. The proof is by induction on the program execution steps, i.e. *step*(u,i), and is divided into a number of cases corresponding to the different types of vertices. In each case, the argument follows the following general outline:

(i) If $A(u)(i)$ is defined, then program point u executes at least i times. From the properties observed earlier, $A(u)(i)$ is shown to be some function f_u of the values computed at some other program points at particular instances:

$$A(u)(i) = f_u(A_{u_1}(1..i_1), A_{u_2}(1..i_2), \ldots)$$

where $step(u_j,i_j) < step(u,i)$ for all j.

(ii) From the inductive hypothesis, we assume the existence of a k such that $S^k(u_j)(1..i_j)$ is defined and equal to $A(u_j)(1..i_j)$, for all j.

(iii) We then look at the definition of $S^{k+1}(u)$, obtained from the set of recursive equations,

$$S^{k+1}(u) = F_u(S^k(v_1), S^k(v_2), \ldots)$$

and show that $S^{k+1}(u)(i)$ is defined and equal to $f_u(A_{u_1}(1..i_1), A_{u_2}(1..i_2), \ldots)$, completing the proof.

Case 1: Let u be the *entry* vertex or some *Initialize* vertex.

This is the base case, and the proof is trivial. Under an appropriate interpretation of these vertices, u executes only once. From the definition we can easily verify that $S^1(u)(1)$ is defined and equal to $A(u)(1)$.

Case 2: Let u be a *FinalUse* vertex.

Let v be its sole reaching definition. Both u and v can execute at most one time, and v must execute before u. The result follows trivially.

Case 3: Let u be a ϕ_T or ϕ_F vertex.

Assume, without loss of generality, that u is a ϕ_T vertex. Let v denote *parent*(u) and w denote *dataPred*(u). From property 11 in the previous section, $j = index(A(v),i,true)$ must be defined and

$$step(w,j) < step(v,j) < step(u,i) < step(w,j+1)$$

and $A(u)(i)$ must be equal to $A(w)(j)$. From the inductive hypothesis, there exists a k such that $S^k(w)(1..j)$ is defined and equal to $A(w)(1..j)$ and $S^k(v)(1..j)$ is defined and equal to $A(v)(1..j)$ (and in particular, $index(S^k(v),i,true) = j$). By definition,

$$S^{k+1}(u) = \text{select}(true, S^k(v), S^k(w))$$

It is a property of *select* that $S^{k+1}(u)(i)$ is defined and equal to $A(u)(i)$.

Case 4: Let u be a ϕ_{if} vertex.

Let v be *ifNode*(u), x be *trueDef*(u) and y be *falseDef*(u). Obviously, the *parent* of both x and y is v. As observed in the previous section, $step(v,i) < step(u,i)$ (property 8), $step(x,j) < step(u,i)$ (property 12), and $step(y,i-j) < step(u,i)$ (property 12), where $j = \#(A(v),i,true)$ and $i-j = \#(A(v),i,false)$. Further, from property 13,

$$A(u)(i) = \textbf{if } A(v)(i) \textbf{ then } A(x)(j) \textbf{ else } A(y)(i-j)$$

From the inductive hypothesis, there exists a k such that $S^k(v)(1..i) = A(v)(1..i)$, $S^k(x)(1..j) = A(x)(1..j)$ and $S^k(y)(1..i-j) = A(y)(1..i-j)$, while from definition,

$$S^{k+1}(u) = merge(S^k(v), S^k(x), S^k(y))$$

It follows that $S^{k+1}(u)(i)$ is defined and equal to $A(u)(i)$, as required.

Case 5: Let u be a ϕ_{exit} or ϕ_{while} vertex.

As can be seen from the defining equations in these cases, these are similar to ϕ_F and ϕ_T vertices, and the proof is similar too.

Case 6: Let u be a ϕ_{enter} vertex.

Let v be *whileNode*(u), x and y be outerDef(u) and innerDef(u) respectively. Let w be the parent of x and v. Assume, without loss of generality, that the control dependences $w \longrightarrow_c v$ and $w \longrightarrow_c u$ are labelled *true*. Consider the case i=1 first. We showed in the previous section (property 14) that *step*(x,1) < *step*(u,1) and that A(u)(1), if defined, must be equal to A(x)(1). Consider $i > 1$. Again, we showed that *step*(v,i-1) < *step*(u,i), *step*(x,j) < *step*(u,i) and *step*(y,i-j) < *step*(u,i) where j = #(A(v),i-1,*false*)+1. Further,

A(u)(i) = **if** A(v)(i-1) **then** A(y)(i-j) **else** A(x)(j)

The hypothesis implies the existence of a k such that $S^k(v)(1..i\text{-}1)$ = A(v)(1..i-1), $S^k(y)(1..i\text{-}j)$ = A(y)(1..i-j) and $S^k(x)(1..j)$ = A(x)(1..j). By definition,

$S^{k+1}(u)$ = *whileMerge*($S^k(v)$, $S^k(y)$, $S^k(x)$)

The properties of *whileMerge* imply that $S^{k+1}(u)(i)$ is defined and equal to A(u)(i).

Case 7: Let u be a ϕ_{copy} vertex.

The proof is similar to the above one, simplified by the fact that there is no definition of *varOf*(u) inside the loop. Let v denote *whileNode*(u), and w *dataPred*(u). We showed in the previous section (property 15) that *step*(v,i-1) < *step*(u,i), *step*(w,j) < *step*(u,i) where j = #(A(v),i-1,*false*)+1, and that A(u)(i) must be equal to A(w)(j). From the hypothesis, there exists a k such that

$S^k(v)(1..i\text{-}1)$ = A(v)(1..i-1) and

$S^k(w)(1..j)$ = A(w)(1..j)

and by definition

$S^{k+1}(u)$ = whileCopy($S^k(v)$, $S^k(w)$)

It follows that $S^{k+1}(u)(i)$ is defined and equal to A(u)(i), as required.

Case 8: Let u be an *assignment* statement, *if* predicate or *while* predicate and let u have at least one data dependence predecessor.

Let u_1, u_2, ... u_n represent the n data dependence predecessors of u. We know that *step*(u_j,i) < *step*(u,i) for all $j \le n$ (property 8) and that A(u)(i) must be equal to *functionOf* (u)(A(u_1)(i_1), ..., A(u_n)(i_n)) (property 9). From the inductive hypothesis, there exists a k such that, for $1 \le j \le n$,

$S^k(u_j)(1..i)$ = A(u_j)(1..i)

From definition,

$S^{k+1}(u)$ = map (*functionOf* (u))($S^k(u_1)$, ..., $S^k(u_n)$)

It follows that $S^{k+1}(u)(i)$ is defined and equal to A(u)(i).

Case 9: Let u be a constant-valued *assignment* statement or *if* predicate.

Let v be u's parent. Assume, without loss of generality, that the control dependence $v \longrightarrow_c u$ is labelled *true*. We know from property 10 of the previous section that j = *Index*(A(v),i,*true*) must be defined and that

step(v,j) < *step*(u,i)

Hence, there exists a k such that $S^k(v)(1..j)$ is defined and equal to A(v)(1..j). By definition,

$S^{k+1}(u)$ = replace(*true*, c, $S^k(v)$)

and the required result follows.

Case 10: Let u be a constant-valued *while* predicate.

If the constant is *false*, the vertex behaves just like vertices in the previous case. If the constant is *true*, and if u executes at least once, then there must be a k and j such that $S^k(v)(j)$ is defined and the same as *label*(v,u), where v is u's parent. From the definition, it can be seen that $S^{k+1}(u)$ is an infinite sequence of

*true*s, satisfying the requirement.

We have proved the lemma for each possible value of *typeOf*(u), and hence the lemma follows. □

THEOREM. *Let G be the extended PRG of a program P and* σ *an input store. Let S(u) denote* **M***[G](*σ*)(u) and A(u) denote the sequence of values computed at program point u for input* σ *under the standard operational semantics. Then, A(u) is a prefix of S(u).*

PROOF. The theorem follows immediately from the previous lemma. Let $S^k(u)$ denote the k^{th} approximation to S(u). Thus, $S(u) = \bigsqcup_{i=0}^{\infty} S^i(u)$. If A($u$)($i$) is defined, then there exists a k such that $S^k(u)(i)$ is defined and equal to A(u)(i), from the previous lemma. Consequently, S(u)(i) is defined and equal to A(u)(i). □

The preceding theorem concerns possibly nonterminating (or abnormally terminating) executions of the program. We now consider executions that terminate normally and show the stronger result that for all program points u, S(u) = A(u).

LEMMA. *Let G be the extended PRG of a program P and* σ *an input store on which P terminates normally. Let S(u) denote* **M***[G][*σ*](u), $S^k(u)$ denote the k^{th} approximation to S(u) and A(u) denote the sequence of values computed at program point u for input* σ *under the standard operational semantics. For any k,* $S^k(u)$ *is a prefix of A(u).*

PROOF. The proof is by induction on k. Assume that the program terminates normally and that $S^k(u)(i)$ is defined. We show that A(u)(i) is defined. The equality of A(u)(i) and $S^k(u)(i)$ then follows from the previous lemma and the fact that $S^k(u)$ is monotonic in k.

Now, A(u)(i) is defined iff u executes i times. Thus, it is enough to show that u executes i times, which we do below. (Similarly, the inductive hypothesis may be interpreted as: if $S^{k-1}(v)(j)$ is defined, then A(v)(j) is defined and, hence, v must have executed j times.)

Case 1: Let u be the *entry* vertex or some *Initialize* vertex.

The proof is trivial in this case.

Case 2: Let u be a *FinalUse* vertex.

Let v denote *dataPred*(u). By definition, $S^k(u) = S^{k-1}(v)$. Thus, if $S^k(u)(i)$ is defined, then so is $S^{k-1}(v)(i)$. From the inductive hypothesis, program point v must have executed i times (which also means that i must be 1, but that is immaterial). Since u and v have the same control dependence predecessors, u must also execute i times (before the program can terminate normally).

Case 3: Let u be a ϕ_T vertex.

Let v denote *ifNode*(u) and w denote *dataPred*(u). By definition, $S^k(u)$ = select(*true*,$S^{k-1}(v)$,$S^{k-1}(w)$) Hence, if $S^k(u)(i)$ is defined, then $S^{k-1}(v)$ must contain at least i *true* values. The hypothesis implies that v must have evaluated to *true* at least i times. Hence u must execute for an i^{th} time. The proof is similar for a ϕ_F vertex.

Case 4: Let u be a ϕ_{if} vertex.

Let w,x, and y denote *ifNode*(u), *trueDef*(u) and *falseDef*(u) respectively. Then, $S^k(u)$ = merge($S^{k-1}(w)$, $S^{k-1}(x)$, $S^{k-1}(y)$). If $S^k(u)(i)$ is defined, then $S^{k-1}(w)(i)$ must also be defined. The hypothesis implies that w must have executed i times. Consequently, u must also have executed i times.

Case 5: Let u be a ϕ_{exit} vertex.

Let v and w denote *whileNode*(u) and *dataPred*(u) respectively. Then, $S^k(u)$ = select(*false*, $S^{k-1}(v)$, $S^{k-1}(w)$). If $S^k(u)(i)$ is defined, then $S^{k-1}(v)$ must contain at least i occurrences of *false*. From the inductive hypothesis, the corresponding *while* loop must have completed execution at least i times. Hence u must have executed at least i times.

Case 6: Let u be a ϕ_{while} vertex.

The proof is similar to the case of a ϕ_T vertex.

Case 7: Let u be a ϕ_{enter} vertex.

Let v,y, and x denote *whileNode*(u), *innerDef*(u) and *outerDef*(u) respectively. Then, $S^k(u)$ = whileMerge($S^{k-1}(v)$, $S^{k-1}(y)$, $S^{k-1}(x)$). Consider the case $i = 1$. If $S^k(u)(1)$ is defined, then $S^{k-1}(x)(1)$ must be defined too. Hence, x must have executed at least once, from the induction hypothesis. Consequently, u must have executed at least once too. Consider the case $i > 1$. If $S^k(u)(i)$ is defined, $S^{k-1}(v)(1..i\text{-}1)$ must be defined too. Consequently, v must have executed i-1 times, by the induction hypothesis. Suppose it evaluated to *true* in the $i-1^{th}$ time, i.e., assume $S^{k-1}(v)(i\text{-}1)$ were *true*. Then u must subsequently execute, for an i^{th} time. On the other hand, let $S^{k-1}(v)(i\text{-}1)$ be false. Let j = #($S^{k-1}(v)$,i-1,false). Then, $S^{k-1}(x)(j+1)$ must be defined. That is, x must have executed at least once after u had executed i-1 times. Hence, u must execute for an i^{th} time too.

Case 8: Let u be a ϕ_{copy} vertex.

The proof is just as in the previous case.

Case 9: Let u be an *assignment*, *if* predicate or *while* predicate, with n data dependence predecessors $u_1 ... u_n$, where $n > 0$.

Then, $S^k(u)$ = map (f)($S^{k-1}(u_1)$, ..., $S^{k-1}(u_n)$). If $S^k(u)(i)$ is defined, then $S^{k-1}(u_j)(i)$ must be defined, for all j. Thus, u_j must have executed i times. Hence, u must also execute i times, since u and all the u_j have the same control dependence predecessors.

Case 10: Let u be a constant valued assignment statement or *if* predicate.

Let v be u's parent. Assume, without loss of generality, that the control dependence $v \longrightarrow_c u$ is labelled *true*. Then $S^k(u)$ = replace(*true*, *functionOf*(u), $S^{k-1}(v)$). Thus, if $S^k(u)(i)$ is defined, then $S^{k-1}(v)$ must contain at least i occurrences of *true*. Hence, v must have evaluated to *true* at least i times. So, u must execute at least i times.

Case 11: Let u be a constant-valued *while* predicate.

If the constant is *false*, the vertex behaves like the vertices in the previous case. Otherwise, if $S^k(u)(i)$ is defined, then its parent v must have evaluated to *label*(v,u) at least once, which would have caused u to execute. This would have resulted in an infinite loop, contradicting the assumption that the program halts. Hence $S^k(u)$ must be a null sequence, for any k, completing the proof. □

THEOREM. *Let G be the extended PRG of a program P and* σ *an input store on which P terminates normally. Let S(u) denote* **M***[G][*σ*](u) and A(u) denote the sequence of values computed at program point u for input* σ *under the standard operational semantics. Then S(u) is a prefix of A(u).*

PROOF. Let $S^k(u)$ denote the k^{th} approximation to S(u). If S(u)(i) is defined, then there must be a k such that $S^k(u)(i)$ is defined (and, obviously, equal to S(u)(i)). It follows from the previous lemma that A(u)(i) is defined and equal to S(u)(i). □

THEOREM. *Let G be the extended PRG of a program P and* σ *an input store on which P terminates normally.* **M***[G][*σ*](u) is equal to A(u), the sequence of values computed at program point u for input* σ *under the standard operational semantics.*

PROOF. It follows from the last two theorems that A(u) is a prefix of **M**[G][σ](u) and **M**[G][σ](u) is a prefix of A(u). Hence, A(u) and **M**[G][σ](u) must be equal. □

A stronger form of the equivalence theorem for PRGs [Yang89c] (see Section 3) follows directly from the previous theorems.

THEOREM. *Let P and Q be programs with isomorphic PRGs, R_P and R_Q respectively. Let σ_1 and σ_2 be two states that agree on the imported variables of P and Q. Let x_1 and x_2 be two corresponding vertices of P and Q. Let $A_P(x_1)$ and $A_Q(x_2)$ denote the sequence of values computed at x_1 and x_2, on states σ_1 and σ_2, respectively. Then, either (1) P and Q terminate normally on σ_1 and σ_2, respectively, and $A_P(x_1)$ equals $A_Q(x_2)$, or (2) neither P nor Q terminates normally on σ_1 and σ_2, respectively, and $A_P(x_1)$ is a prefix of $A_Q(x_2)$ or vice versa.*

PROOF. Note that the dependence of the PRG semantics on the initial state is restricted to the values of the imported variables. Consequently, the semantics of the isomorphic PRGS R_P and R_Q for initial states σ_1 and σ_2, respectively, say S_P and S_Q, are identical. Thus $S_P(x_1) = S_Q(x_2)$. From the previous section, we also know that $A_P(x_1)$ is a prefix of $S_P(x_1)$ and that $A_Q(x_2)$ is a prefix of $S_Q(x_2)$. Consequently, $A_P(x_1)$ must be a prefix of $A_Q(x_2)$ or vice versa.

Note that $A_P(x)$ and $S_P(x)$ are finite for all vertices x in P iff the program P terminates normally. Hence, either P and Q both terminate normally (in which case, $A_P(x_1) = S_P(x_1) = S_Q(x_2) = A_Q(x_2)$) or neither P nor Q terminates normally. The theorem follows immediately. □

However, note that this stronger equivalence theorem can be derived from the Sequence-Congruence Theorem [Yang89a] too.

REFERENCES

Aho86a.
Aho, A.V., Sethi, R., and Ullman, J.D., *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, Reading, MA (1986).

Alpern88a.
Alpern, B., Wegman, M.N., and Zadeck, F.K., "Detecting equality of variables in programs," pp. 1-11 in *Conference Record of the Fifteenth ACM Symposium on Principles of Programming Languages*, (San Diego, CA, January 13-15, 1988), ACM, New York, NY (1988).

Cartwright89a.
Cartwright, R. and Felleisen, M., "The semantics of program dependence," *Proceedings of the ACM SIGPLAN 89 Conference on Programming Language Design and Implementation*, (Portland, OR, June 21-23, 1989), *ACM SIGPLAN Notices*, pp. 13-27 ACM, (June 21-23 1989).

Cytron89a.
Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., and Zadeck, K., "An efficient method of computing static single assignment form," pp. 25-35 in *Conference Record of the Sixteenth ACM Symposium on Principles of Programming Languages*, (Austin, TX, Jan. 11-13, 1989), ACM, New York, NY (1989).

Ferrante87a.
Ferrante, J., Ottenstein, K., and Warren, J., "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems* **9**(3) pp. 319-349 (July 1987).

Horwitz88a.
Horwitz, S., Prins, J., and Reps, T., "On the adequacy of program dependence graphs for representing programs," pp. 146-157 in *Conference Record of the Fifteenth ACM Symposium on Principles of Programming Languages*, (San Diego, CA, January 13-15, 1988), ACM, New York, NY (1988).

Horwitz89a.
Horwitz, S., Prins, J., and Reps, T., "Integrating non-interfering versions of programs," *ACM Trans. Program. Lang. Syst.* **11**(3) pp. 345-387 (July 1989).

Kahn74a.
Kahn, G., "The semantics of a simple language for parallel programming," pp. 471-475 in *IFIP 1974 proceedings*, (1974).

Kuck72a.
Kuck, D.J., Muraoka, Y., and Chen, S.C., "On the number of operations simultaneously executable in FORTRAN-like programs and their resulting speed-up," *IEEE Trans. on Computers* **C-21**(12) pp. 1293-1310 (December 1972).

Kuck81a.
Kuck, D.J., Kuhn, R.H., Leasure, B., Padua, D.A., and Wolfe, M., "Dependence graphs and compiler optimizations," pp. 207-218 in *Conference Record of the Eighth ACM Symposium on Principles of Programming Languages*, (Williamsburg, VA, January

26-28, 1981), ACM, New York, NY (1981).

Reps89a.
Reps, T. and Yang, W., "The semantics of program slicing and program integration," pp. 360-374 in *Proceedings of the Colloquium on Current Issues in Programming Languages*, (Barcelona, Spain, March 13-17, 1989), *Lecture Notes in Computer Science*, Vol. 352, Springer-Verlag, New York, NY (1989).

Rosen88a.
Rosen, B., Wegman, M.N., and Zadeck, F.K., "Global value numbers and redundant computations," pp. 12-27 in *Conference Record of the Fifteenth ACM Symposium on Principles of Programming Languages*, (San Diego, CA, January 13-15, 1988), ACM, New York, NY (1988).

Selke89a.
Selke, R.P., "A rewriting semantics for program dependence graphs," *Conference Record of the Sixteenth ACM Symposium on Principles of Programming Languages*, pp. 12-24 ACM, (Jan.11-13,1989).

Shapiro70a.
Shapiro, R.M. and Saint, H., "The representation of algorithms," Technical Report CA-7002-1432, Massachusetts Computer Associates (February 1970). As cited in [Alpern88, Rosen88].

Yang89a.
Yang, W., Horwitz, S., and Reps, T., "Detecting program components with equivalent behaviors," TR-840, Computer Sciences Dept., Univ. of Wisconsin, Madison, WI. (April 1989).

Yang89b.
Yang, W., Horwitz, S., and Reps, T., "A new program integration algorithm," Technical Report in preparation, Computer Sciences Department, University of Wisconsin, Madison, WI (September 1989).

Yang89c.
Yang, W., Horwitz, S., and Reps, T., "The Semantic Properties of Program Representation Graphs," Technical Report in preparation, Computer Sciences Dept., Univ. of Wisconsin, Madison, WI. (Summer 1989).