On the Adequacy of Program Dependence Graphs for Representing Programs

Susan Horwitz, Jan Prins, and Thomas Reps
University of Wisconsin – Madison

Abstract

Program dependence graphs were introduced by Kuck as an intermediate program representation well suited for performing optimizations, vectorization, and parallelization. There are also additional applications for them as an internal program representation in program development environments.

In this paper we examine the issue of whether a program dependence graph is an adequate structure for representing a program's execution behavior. (This question has apparently never been addressed before in the literature). We answer the question in the affirmative by showing that if the program dependence graphs of two programs are isomorphic then the programs are strongly equivalent.

1. Introduction

Program dependence graphs were introduced by Kuck as an intermediate program representation well suited for performing optimizations, vectorization, and parallelization [8,12,9,10]. A number of variations have since been discussed [25]. Additional applications for program dependence graphs are as the internal structure for representing programs in a language-based program development environment [11] as well as for integrating program variants and determining whether enhancements made to different program versions interfere [7].

Although there exists an extensive body of work that makes use of program dependence graphs, we were unable to find any published proof that program dependence graphs were "adequate" as a program representation. One would like a proof that program dependence graphs distinguish between inequivalent programs; that is, two inequivalent programs should have different program dependence graphs. Both Ken Kennedy and Jeanne Ferrante acknowledged that they did not know where such a proof could be found [private communication, Jan. 1987].

In this paper, we prove that for a language with assignment statements, conditional statements, and while-loops, a program dependence graph does capture a program's behavior. The concept of "programs with the same behavior" is formalized as the concept of *strong equivalence*: two programs are strongly equivalent iff, for any initial state σ, either both programs diverge or both halt with the same final state. We prove a theorem, the Equivalence Theorem, that states that if the program dependence graphs of two programs are isomorphic then the programs are strongly equivalent.

We also show that the program dependence representation used here (a somewhat nonstandard representation) has a "minimality" property; omitting any class of data-dependency edges used in the representation permits inequivalent programs to have isomorphic program dependence graphs.

Although the language for which we prove the Equivalence Theorem does include array variables, when computing data dependencies we treat an assignment to an array element as a conditional assignment to the entire array and a reference to an array element as a reference to the entire array. This is a more simplistic approach than has been taken in most previous work that uses program dependence representations; such work usually includes analysis of array index expressions [4,13,2,3]. To provide justification for this work, our definition of data dependencies would have to incorporate these analysis techniques, and our proof would have to be modified to cover the extended definition.

It is worthwhile to review the value of the Equivalence Theorem. In some of the work in which program dependence representations are used for program optimization, they have been employed in a rather restricted fashion, as an auxiliary data structure for discovering optimizing transformations. In PFC [2], for example, the internal program representation consists of a control-flow graph augmented with a program dependence representation; both structures are updated as a program is transformed. The Equivalence Theorem assures that, *by itself*, a properly defined program dependence graph is a suitable structure from which to discover and perform optimiza-

---

This work was supported in part by the National Science Foundation under grants DCR-8552602 and DCR-8603356 as well as by grants from IBM, DEC, Siemens, and Xerox.

Authors' current addresses: Susan Horwitz and Thomas Reps, Computer Sciences Department, Univ. of Wisconsin, 1210 W. Dayton St., Madison, WI 53706; Jan Prins, Dept. of Computer Science, Sitterson Hall 083a, Univ. of North Carolina, Chapel Hill, NC 27514.

tions.

The Equivalence Theorem demonstrates that it would make sense to give a semantics for the *feasible* program dependence graphs – those that are the program dependence graph of some program. The theorem assures that the program dependence graph would be a suitable structure for direct interpretation, as has been proposed as one of their uses in a programming environment [11]. The theorem also assures that the program dependence graph is a suitable structure from which to generate machine code.

In [7], an algorithm is presented for integrating several variants of a base program (or determining that the variants incorporate interfering changes). In the algorithm, program dependence graphs are used to determine what changes in behavior should be preserved in the integrated program. The integrated program is created by (1) merging the program dependence graphs for the base and variant programs, (2) testing the merged dependence graph for interference conditions, and (3) reconstituting a program from the merged dependence graph. The Equivalence Theorem assures that all programs that could be created from the merged program dependence graph are strongly equivalent.

The assumption of the Equivalence Theorem is that programs P and Q have isomorphic program dependence graphs, and the argument used in the proof involves showing (roughly) that a subtree T of program P is strongly equivalent to the subtree U of program Q whose components are isomorphic to T's components. The theorem is proved by structural induction over the abstract syntax of the programming language; the induction hypothesis is that each subtree T_i of T is strongly equivalent to a corresponding subtree U_i of U.

The crux of the proof is showing the necessary equivalence for statement lists. In this case, T and U are two sequences of corresponding (but not identical) components, where the two sequences are permutations of one another. Because the two sequences are permutations of one another, their initial subsequences are not equivalent, and we were unable to formulate a proof by induction on the length of one sequence. Instead, we use a kind of reduction step. We first introduce an extended language $\tilde{L}$, in which only straight-line code is permitted, then prove a lemma, the Block-Equivalence Lemma, which is essentially the Equivalence Theorem for straight-line code; it says that if the program dependence graphs of two $\tilde{L}$ programs are isomorphic then the programs are equivalent.

The reduction step consists of a "semantic flattening" in which code sequences from the original language (call it L), are translated into (straight-line) sequences in $\tilde{L}$. The term "semantic flattening" for this translation is suggestive because an expression on the right-hand side of an assignment statement in L may contain an application of one of the meaning functions for language L to a construct of L and to a state. By this device, an entire subtree of sequence $\tilde{T}$ gets "flattened" by the translation into a collection of $\tilde{L}$ assignment statements.

The proof proceeds by showing (1) that the translations of non-straight-line sequences T and U in L to straight-line sequences $\tilde{T}$ and $\tilde{U}$ in $\tilde{L}$ preserve meaning, and (2) that $\tilde{T}$ and $\tilde{U}$ have the same program dependence graph, and hence are equivalent by the Block-Equivalence Lemma. We conclude that T and U are equivalent, which permits us to push through an inductive argument for the Equivalence Theorem.

It is the "semantic-flattening" operation together with the Block-Equivalence Lemma that allow us to overcome the difficulty alluded to earlier, namely that the components that make up U are a permutation of the components that make up T. Semantic flattening is an interesting idea in its own right and may prove useful in other contexts.

The rest of this paper is organized into four sections. Section 2 introduces terminology and notation, including the definition of *program dependence graphs*, the program dependence representation that we use. Section 3 presents the proof of the Equivalence Theorem. Section 4 discusses a minimality property of program dependence graphs. Section 5 compares program dependence graphs to more standard program dependence representations, and argues that the Equivalence Theorem applies to the latter as well.

2. Terminology and Notation

2.1. Equivalence of Programs

This paper is concerned with showing that program dependence graphs partially characterize programs that have the same behavior. The concept of "programs with the same behavior" is formalized as the concept of *strong equivalence*, defined as follows:

Definition. Two programs P and Q are *strongly equivalent* iff for any state σ, either P and Q both diverge when initiated on σ or they both halt with the same final values for all variables. If P and Q are not strongly equivalent, we say they are *inequivalent*.

We use the term "divergence" to mean both nontermination (for example, because of infinite loops) and abnormal termination (for example, because of division by zero or the use of an out-of-bounds array index).

2.2. Abstract Syntax

We are concerned with a programming language that has assignment statements, conditional statements, and while-loops, and whose expressions contain scalar and array variables and constants. The abstract syntax of the language is defined as the terms of the types *lvalue*, *exp*,

stmt, *stmt_list*, and *program* constructed using the operators *Assign*, *While*, *IfThenElse*, *StmtList*, and *Program*. The five operators of the abstract syntax have the following definitions:

Assign : $lvalue \times exp \rightarrow stmt$
While : $exp \times stmt\_list \rightarrow stmt$
IfThenElse : $exp \times stmt\_list \times stmt\_list \rightarrow stmt$
StmtList : $stmt \times stmt \times \ldots \times stmt \rightarrow stmt\_list$
Program : $stmt\_list \rightarrow program$

Henceforth, we use "program" and "abstract syntax tree" synonymously.

2.3. Program Dependence Graphs

Different definitions of program dependence representations have been given, depending on the intended application; nevertheless, they are all variations on a theme introduced in [8], and share the common feature of having explicit representations of both control dependencies and data dependencies. The representation used in this paper is the *program dependence graph* defined below. Its features are somewhat non-standard: ordinarily, two kinds of data dependencies, called *anti-dependencies* and *output dependencies* are used in addition to flow dependencies; in our representation, we omit anti-dependencies and replace output dependencies with *def-order* dependencies. (Section 4 discusses the differences between output and def-order dependencies).

The program dependence graph for a program P, denoted by G_P, is a directed graph whose vertices are connected by several kinds of edges.[1] The vertices of G_P represent the assignment statements and control predicates that occur in program P. In addition, G_P includes three other categories of vertices:

a) There is a distinguished vertex called the *entry vertex*.

b) For each variable x used in P, there is a vertex called the *initial definition of x*. This vertex represents an initial assignment to x where the value of x is retrieved from the initial state.

c) For each variable used in P, there is a second vertex called the *final use of x*. It represents an access to the final value of x computed by P.

The edges of G_P represent *dependencies* among program components. An edge represents either a *control dependency* or a *data dependency*. Control dependency edges are labeled either **true** or **false**, and the source of a control dependency edge is always the entry vertex or a predicate vertex. A control dependency edge from vertex v_1 to vertex v_2, denoted $v_1 \longrightarrow_c v_2$, means that during execution, whenever the predicate represented by v_1 is evaluated and its value matches the label on the edge to v_2, then the program component represented by v_2 will be executed (although perhaps not immediately). A method for determining control dependency edges for arbitrary programs is given in [5]; however, because we are assuming that programs include only assignment, conditional, and while statements, the control dependency edges of G_P can be determined in a much simpler fashion. For the language under consideration here, a program dependence graph contains a *control dependency edge* from vertex v_1 to vertex v_2 of G_P iff one of the following holds:

i) v_1 is the entry vertex, and v_2 represents a component of P that is not subordinate to any control predicate. The edge $v_1 \longrightarrow_c v_2$ is labeled **true**.

ii) v_1 represents a control predicate, and v_2 represents a component of P immediately subordinate to the control construct whose predicate is represented by v_1. If v_1 is the predicate of a while-loop, the edge $v_1 \longrightarrow_c v_2$ is labeled **true**; if v_1 is the predicate of a conditional statement, the edge $v_1 \longrightarrow_c v_2$ is labeled **true** or **false** according to whether v_2 occurs in the **then** branch or the **else** branch, respectively.

In other definitions that have been given for control dependency edges, there is an additional edge for each predicate of a **while** statement – each predicate has an edge to itself labeled **true**. By including the additional edge, the predicate's outgoing **true** edges consist of every program element that is guaranteed to be executed (eventually) when the predicate evaluates to **true**. This kind of edge is left out of our definition because it is not necessary for our purposes.

A data dependency edge from vertex v_1 to vertex v_2 means that the program's computation might be changed if the relative order of the components represented by v_1 and v_2 were reversed. In this paper, program dependence graphs contain two kinds of data-dependency edges, representing *flow dependencies* and *def-order dependencies*.

A program dependence graph contains a flow dependency edge from vertex v_1 to vertex v_2 iff i), ii) and either iiia) or iiib) below all hold:

i) Variable x is a scalar and v_1 is a vertex that defines x, or variable x is an array and v_1 is a vertex that defines some element of x.

ii) Variable x is a scalar and v_2 is a vertex that uses x, or variable x is an array and v_1 is a vertex that uses some element of x.

iiia) (Scalar case)
Variable x is a scalar. Control can reach v_2 after v_1

[1] A *directed graph* G consists of a set of *vertices* $V(G)$ and a set of *edges* $E(G)$, where $E(G) \subseteq V(G) \times V(G)$. Each edge $(b, c) \in E(G)$ is directed from b to c; we say that b is the *source* and c the *target* of the edge. Throughout the paper, the term "vertex" is used to refer to elements of dependency graphs, whereas the term "node" refers to elements of derivation trees.

via an execution path along which there is no intervening definition of x. That is, there is a path in the standard control-flow graph for the program [1] by which the definition of x at v_1 reaches the use of x at v_2. (Initial definitions of variables are considered to occur at the beginning of the control-flow graph, and final uses of variables are considered to occur at its end).

iiib) (Array case)
Variable x is an array. Control can reach v_2 after v_1 (*i.e.* an assignment to an array element is treated like a conditional assignment to the entire array, and a use of an array element is treated like a use of the entire array).

A flow dependency that exists from vertex v_1 to vertex v_2 will be denoted by $v_1 \longrightarrow_f v_2$.

Flow dependencies can be further classified as *loop independent* or *loop carried*. A flow dependency $v_1 \longrightarrow_f v_2$ is carried by loop L, denoted by $v_1 \longrightarrow_{lc(L)} v_2$, if in addition to i), ii), and iii) above, the following also hold:

iv) There is an execution path that both satisfies the conditions of iii) above and includes a backedge to the predicate of loop L; and

v) Both v_1 and v_2 are enclosed in loop L.

A flow dependency $v_1 \longrightarrow_f v_2$ is loop independent if in addition to i), ii), and iii) above, there is an execution path that satisfies iii) above and includes *no* backedge to the predicate of a loop that encloses both v_1 and v_2. It is possible to have both $v_1 \longrightarrow_{lc(L)} v_2$ and $v_1 \longrightarrow_{li} v_2$.

A program dependence graph contains a def-order dependency edge from vertex v_1 to vertex v_2 iff all of the following hold:

i) v_1 and v_2 both define the same variable.

ii) v_1 and v_2 are in the same branch of any conditional statement that encloses both of them.

iii) There exists a program component v_3 such that $v_1 \longrightarrow_f v_3$ and $v_2 \longrightarrow_f v_3$.

iv) v_1 occurs to the left of v_2 in the program's abstract syntax tree.

A def-order dependency from v_1 to v_2 is denoted by $v_1 \longrightarrow_{do(v_3)} v_2$.

Note that a program dependence graph is a multi-graph (*i.e.* it may have more than one edge of a given kind between two vertices). When there is more than one loop-carried flow dependency edge between two vertices, each is labeled by a different loop that carries the dependency. When there is more than one def-order edge between two vertices, each is labeled by a vertex that is flow-dependent on both the definition that occurs at the edge's source and the definition that occurs at the edge's target.

The data-dependency edges of a program dependence graph are computed using data-flow analysis. For the restricted language considered in this paper, the necessary computations can be defined in a syntax-directed manner [6].

2.4. Corresponding Subtrees

The subgraph induced by the *control* dependencies of program dependence graph G_P forms a tree that is closely related to the abstract syntax tree for program P. The control dependence subtree is rooted at the entry vertex of G_P, which corresponds to the Program node at the root of P's abstract syntax tree. Each predicate vertex v of G_P corresponds to an interior node of the abstract syntax tree; the node is a While node or an IfThenElse node depending on whether v is labeled with **while** or **if**, respectively. Each assignment vertex of G_P corresponds to an Assign node of the abstract syntax tree.

Because a given program dependence graph G_P has a unique control dependence subgraph, the programs with program dependence graphs isomorphic to G_P are a subset of the programs obtained by permuting the statements subordinate to P's StmtList operators. If P and Q are two programs that have the same program dependence graph, there is natural correspondence between subtrees in P and subtrees in Q, defined as follows:

Definition. Suppose that P and Q are two programs that have the same program dependence graph. Then for each subtree T of P, the subtree of Q that consists of exactly the components that occur in T is said to *correspond* to T.

For each subtree T of P, there is always a corresponding subtree of Q. If T corresponds to U, each subtree of T corresponds to a subtree of U, and *vice versa*; however, the order in which the subtrees of U occur may be a permutation of the order of the corresponding subtrees in T.

3. The Equivalence Theorem

We now address the relationship between a program's program dependence graph and the program's execution behavior. In particular, we show that if the program dependence graphs of two programs are isomorphic then the programs are strongly equivalent. We use the symbol "$\approx$" to denote isomorphism. (For brevity, we occasionally speak of "programs with the *same* program dependence graphs" and "program dependence graphs with *identical* components." These should be understood to mean "corresponding under the isomorphism").

The main result of the paper is the following theorem:

THEOREM. (EQUIVALENCE THEOREM). *If P and Q are programs for which $G_P \approx G_Q$, then P and Q are strongly equivalent.*

Restated in the contrapositive the theorem reads: Inequivalent programs have non-isomorphic program dependence graphs.

3.1. Relativized Strong Equivalence

Our ultimate goal is to use structural induction to show that *programs P* and *Q* with isomorphic program dependence graphs are strongly equivalent; however, corresponding *subtrees T* of *P* and *U* of *Q* may not be strongly equivalent. To handle equivalence of subtrees properly we must generalize the concept of "strongly equivalent programs" to that of "subtrees that are strongly equivalent relative to an input set of variables and an output set of variables."

Definition. Two subtrees, *T* and *U*, are *strongly equivalent relative to an input set of variables In and an output set of variables Out* iff for all states σ and σ' that agree on *In*, either *P* and *Q* both diverge when initiated on σ and σ', respectively, or they both halt with the same final values for all variables in *Out*.

We can use relativized strong equivalence for the induction steps of the proof of the Equivalence Theorem if we choose the sets *In* and *Out* so that for programs, as opposed to subtrees of programs, strong equivalence is the same as relativized strong equivalence. This motivates the following definitions of a (sub)tree's *imported* and *exported* variables:

Definition. The *outgoing flow edges* of a subtree *T* consist of all the loop-independent flow edges whose source is in *T* but whose target is not in *T*, together with all the loop-carried flow edges for which the source is in *T* and the edge is carried by a loop that encloses *T*. Note that the target of an outgoing loop-carried flow edge may or may not be in *T*. The variables *exported* from a subtree *T* are the variables defined at the source of an outgoing flow edge. When the variable defined at the source of an outgoing flow edge is an array element, the *entire* array is exported.

Definition. The *incoming flow edges* of a subtree *T* consist of all the loop-independent flow edges whose target is in *T* but whose source is not in *T*, together with all the loop-carried flow edges for which the target is in *T* and the edge is carried by a loop that encloses *T*. Note that the source of an incoming loop-carried flow edge may or may not be in *T*. The *incoming def-order edges* of a subtree *T* consist of all the def-order edges whose target is in *T* but whose source is not in *T*. The variables *imported* by a subtree *T* are the variables defined at the source of an incoming flow edge or at the source of an incoming def-order edge. When the variable defined at the source of an incoming flow or def-order edge is an array element, the *entire* array is imported.

There are loop-independent flow edges to all final-use vertices of a program dependence graph; thus, the exported variables of a program *P* consist of *all* variables that occur in *P*. The imported variables of a program *P* consist of those variables that may get their values from the initial state. This may be a proper subset of the variables that occur in *P*; however, if two subtrees are strongly equivalent relative to input set *In* and output set *Out*, they are also strongly equivalent relative to input set $In' \supseteq In$ and output set *Out*. Thus, if *programs P* and *Q* are strongly equivalent relative to their imported and exported variables, *P* and *Q* are strongly equivalent.

3.2. The Equivalence Lemma and the Self-Equivalence Lemma

We now state the main lemma needed to prove the Equivalence Theorem.

LEMMA. (EQUIVALENCE LEMMA). *Suppose that P and Q are programs for which* $G_P \approx G_Q$. *Then for any subtrees T in P and U in Q that correspond, T and U are strongly equivalent relative to their imported and exported variables*.

The proof of the Equivalence Lemma is by structural induction on the abstract syntax of the programming language. The proof is straightforward for the operators Assign, While, and IfThenElse; as stated in the Introduction, the StmtList operator is problematic. To handle the StmtList operator we use a "semantic flattening" operation by which the components of StmtLists *T* and *U* are translated into a language that allows only straight-line code. To show that these translations preserve meaning and that the resulting straight-line code sequences are equivalent, we use two lemmas: the Self-Equivalence Lemma, stated and proved below, and the Block-Equivalence Lemma, stated and proved in Section 3.3.

The Self-Equivalence Lemma shows that the definitions of imported and exported variables are consistent with each other and can be used to characterize the state transforming properties of a subtree. In Section 3.4 in the proof of the StmtList case of the Equivalence Lemma, we rely on the Self-Equivalence Lemma to show that the semantic flattening operation used in this case is meaning preserving.

LEMMA. (SELF-EQUIVALENCE LEMMA). *Let T be a subtree of program P. Then T is strongly equivalent to T relative to T's imported and exported variables (as defined in the context given by P)*.

PROOF. The proof is by structural induction on the abstract syntax of the programming language. The proof splits into five cases based on the abstract-syntax operator that appears at the root of *T*.

Throughout the proof, we use σ_1 and σ_1' to denote states that agree on *T*'s imported variables, Imp_T. We use σ_i to denote a sequence of states in the execution of *T* initiated on σ_1, and we use σ_i' to denote the corresponding sequence of states in the execution of *T* initiated on σ_1'.

Case 1. The operator at the root of T is the Assign operator. Suppose T assigns to variable x as a function of variables $\{y_j\}$. Then $\{y_j\} \subseteq Imp_T$, and because by assumption σ_1 and σ_2' agree on Imp_T, T either aborts on both σ_1 and σ_2', or terminates normally on both σ_1 and σ_2'. Consider the case when T terminates normally: Imp_T is either $\{y_j\}$ or $\{y_j\}\cup\{x\}$ (Imp_T is $\{y_j\}\cup\{x\}$ when T is the target of a def-order edge; this is always the case when x is an array, and can also occur when x is a scalar). Exp_T is either $\varnothing$ or $\{x\}$. For any combination of these possibilities, σ_2 and σ_2' agree on x, and hence they agree on Exp_T.

Case 2. The operator at the root of T is the While operator. We use Imp_T and Exp_T, Imp_{exp} and Exp_{exp}, and Imp_{stmt_list} and Exp_{stmt_list} to denote the imported and exported variables of T, T's *exp* component, and T's *stmt_list* component, respectively. We use σ_i and σ_i' to denote the execution state before executing the i^{th} iteration of the loop starting from two states that agree on Imp_T, σ_1 and σ_1', respectively.

We need to show that either T diverges on both σ_1 and σ_1' or else both executions halt after the j^{th} iteration in states σ_{j+1} and σ_{j+1}', respectively, where σ_{j+1} and σ_{j+1}' agree on Exp_T. Because for a loop $Exp_T \subseteq Imp_T$,[2] it suffices to show that if σ_i and σ_i' agree on Imp_T then either T terminates in the states σ_i and σ_i' or the i^{th} iteration computes σ_{i+1} and σ_{i+1}' that agree on Imp_T.

First, we show that $Imp_T = Imp_{exp} \cup Imp_{stmt\_list}$. It is clear that we could have written this with $\subseteq$, noting that Imp_{stmt_list} can include a variable x that is used at the target t of a loop-carried flow dependency edge where the dependence is carried by T. However, there then has to exist an incoming loop-independent flow edge to t, which implies that $v \in Imp_T$.

Let σ_i and σ_i' be states that agree on Imp_T. Evaluating T's condition (the *exp* component of T) in σ_i and σ_i' yields the same value. If the condition evaluates to false, then both executions terminate in the states σ_i and σ_i', which agree on Exp_T.

Now suppose the condition evaluates to true. By the induction hypothesis the *stmt_list* is strongly equivalent to itself relative to Imp_{stmt_list} and Exp_{stmt_list}. Because σ_i and σ_i' agree on Imp_{stmt_list}, either both executions of the *stmt_list* diverge or both terminate in states σ_{i+1} and σ_{i+1}' that agree on Exp_{stmt_list}. If σ_{i+1} and σ_{i+1}' do not also agree on Imp_T, then let $x \in Imp_T$ be a variable on which they disagree (so $x \notin Exp_{stmt\_list}$). Now, by assumption, σ_i and σ_i' agree on Imp_T; therefore, at least one of the two executions of *stmt_list* executed an assignment statement a that assigned a value to x and reached the end of the *stmt_list*. There are two cases to consider:

(1) One possibility is that $x \in Imp_T$ because x is used in a statement b that is the target of an incoming flow edge *If this were the case, then there must be a loop-carried flow edge* $a \longrightarrow_{lc(T)} b$. *This implies that* $x \in Exp_{stmt\_list}$, *which contradicts our previous assumption.*

(2) The other possibility is that $x \in Imp_T$ because the *stmt_list* has an incoming def-order edge $\ldots \longrightarrow_{do(c)} d$. However, this implies that there is an outgoing flow edge $a \longrightarrow_f c$ from the *stmt_list*. This implies that $x \in Exp_{stmt\_list}$, which contradicts our previous assumption.

We conclude that σ_{i+1} and σ_{i+1}' agree on Imp_T, and hence T is strongly equivalent to itself relative to Imp_T and Exp_T.

Case 3. The operator at the root of T is the IfThenElse operator. Evaluating T's condition (the *exp* component of T) in σ_1 and σ_1' yields the same value; without loss of generality, assume that the condition evaluates to true.

By the induction hypothesis, the true-branch of T is strongly equivalent to itself relative to its imported variables, Imp_{true}, and its exported variables, Exp_{true}. Thus, when initiated in states σ_1 and σ_1' either the true-branch of T diverges on both or terminates in σ_2 and σ_2', respectively.

Note that $Exp_T = Exp_{true} \cup Exp_{false}$. By the induction hypothesis, σ_2 and σ_2' agree on Exp_{true}. If they do not also agree on Exp_{false}, then let $x \in Exp_{false}$ be a variable on which they disagree (so $x \notin Exp_{true}$). Because $x \in Exp_{false}$, there is an assignment statement a in the false branch of T that assigns to x and is the source of an outgoing flow edge from that branch (say $a \longrightarrow_f b$).

We must consider whether it is possible that $x \notin Imp_T$. By assumption, $x \notin Exp_{true}$; however, there is an execution path from the initial definition of x to b that does not pass through the false branch of T. Let c represent the last definition to x along this path, so $c \longrightarrow_f b$, which implies that $c \longrightarrow_{do(b)} a$. Therefore, it must be that $x \in Imp_T$.

Because $x \in Imp_T$, σ_1 and σ_1' agree on x. Because σ_2 and σ_2' disagree on x, at least one of the two executions of the true branch of T executed an assignment statement d that assigned a value to x and reached the end of the true branch of T. But this implies the existence of a flow edge $d \longrightarrow_f b$, so $x \in Exp_{true}$, which contradicts a previous assumption. We conclude that σ_2 and σ_2' agree on Exp_{false}. This, together with the fact that σ_2 and σ_2' agree on Exp_{true}, means that T is strongly equivalent to T relative to Imp_T and Exp_T.

---

[2] If $x \in Exp_T$, then T contains an assignment a to x with an outgoing flow edge $a \longrightarrow_f b$. Because the loop may execute zero times, the assignment to x must be the target of a def-order edge $\ldots \longrightarrow_{do(b)} a$, hence $x \in Imp_T$.

Case 4. The operator at the root of T is the StmtList operator. Let $T_1, T_2, \cdots, T_n$ denote the immediate subtrees of T. We use σ_i and σ_i' to denote the execution state before executing T_i; we use Imp_i and Exp_i to denote the imported and exported variables, respectively, of T_i; and we use $Imp_{1..i}$ and $Exp_{1..i}$ to denote the imported and exported variables, respectively, of the initial subsequence $T_1, T_2, \cdots, T_i$. (Although the imported and exported variables for subsequences were not part of the definition in Section 3.1, we intend the obvious extension: the imported variables of a subsequence is defined in terms of incoming edges whose targets are inside the subsequence; the exported variables of a subsequence is defined in terms of outgoing edges whose sources are inside the subsequence).

The proof of this case is by induction over the initial subsequences of T. We want to show that for all i, $1 \le i \le n$, $T_1, T_2, \cdots, T_i$ is strongly equivalent to itself relative to $Imp_{1..i}$ and $Exp_{1..i}$.

Base case. $n = 1$. The proposition follows immediately from the induction hypothesis of the structural induction.

Induction step. The induction hypothesis is: If σ_1 and σ_1' agree on $Imp_{1..i}$ then σ_{i+1} and σ_{i+1}' agree on $Exp_{1..i}$. Thus, if σ_1 and σ_1' are arbitrary states that agree on $Imp_{1..i+1}$, we need to show that σ_{i+2} and σ_{i+2}' agree on $Exp_{1..i+1}$.

Note that $Imp_{1..i} \subseteq Imp_{1..i+1}$, which means that σ_1 and σ_1' agree on $Imp_{1..i}$, and thus, by the induction hypothesis, σ_{i+1} and σ_{i+1}' agree on $Exp_{1..i}$.

First, we must show that σ_{i+1} and σ_{i+1}' agree on Imp_{i+1}. Any variable $x \in Imp_{i+1}$ on which σ_{i+1} and σ_{i+1}' disagree must be in $Imp_{1..i+1}$ (if not, x would be in $Exp_{1..i}$ on which σ_{i+1} and σ_{i+1}' agree). By assumption, σ_1 and σ_1' agree on $Imp_{1..i+1}$; consequently, at least one of the two executions performed an assignment, a, that assigned to x and reached the end of T_i. There are now two cases to consider:

(1) One possibility is that $x \in Imp_{i+1}$ because x is used in a statement b that is the target of one of T_{i+1}'s incoming flow edges. In this case, there is a flow edge: $a \longrightarrow_f b$. This implies that $x \in Exp_{1..i}$, so σ_{i+1} and σ_{i+1}' must agree on x, which contradicts our assumption that they disagree on x.

(2) The other possibility is that $x \in Imp_{i+1}$ because there is an incoming def-order edge, $\ldots \longrightarrow_{do(d)} c$, to T_{i+1}. However, this implies that there is an outgoing flow edge of $Exp_{1..i}$: $a \longrightarrow_f d$. As in the previous case, this implies that $x \in Exp_{1..i}$, so σ_{i+1} and σ_{i+1}' must agree on x, which contradicts our assumption that they disagree on x.

Because σ_{i+1} and σ_{i+1}' must agree on Imp_{i+1}, the induction hypothesis of the structural induction implies that the executions of T_{i+1} on σ_{i+1} and σ_{i+1}' either both diverge or both terminate in states σ_{i+2} and σ_{i+2}' that agree on Exp_{i+1}.

The final step is to show that σ_{i+2} and σ_{i+2}' agree on $Exp_{1..i+1}$. Note that $Exp_{1..i+1} \supseteq Exp_{i+1}$. Now suppose there is a variable $x \in Exp_{1..i+1}$ on which σ_{i+2} and σ_{i+2}' disagree (in particular, $x \notin Exp_{i+1}$). By the induction hypothesis, σ_{i+1} and σ_{i+1}' agree on $Exp_{1..i}$, so at least one of the two executions of T_{i+1} performed an assignment, a, that assigned to x and reached the end of T_{i+1}. Because $x \in Exp_{1..i+1}$, there must also be an outgoing flow edge $a \longrightarrow_f \ldots$ from T_{i+1}. This implies that $a \in Exp_{i+1}$, so σ_{i+2} and σ_{i+2}' must agree on x, which contradicts our assumption that they disagree on x.

This completes the induction, so we conclude that T is strongly equivalent to itself relative to Imp_T and Exp_T.

Case 5. The operator at the root of T is the Program operator. Because $Imp_T = Imp_{stmt\_list}$ and $Exp_T = Exp_{stmt\_list}$, the strong equivalence of T with itself relative to Imp_T and Exp_T follows directly from the induction hypothesis. □

3.3. An Extended Language and the Block-Equivalence Lemma

At this point, we introduce a second programming language that is used in the proof of the Equivalence Lemma. We will use L to denote the programming language that has been considered so far; the second language is referred to as $\tilde{L}$. The definition of $\tilde{L}$ is tailored to its use in the proof, and is highly artificial. One feature of $\tilde{L}$ is that only straight-line code is permitted. The device used in the proof of the Equivalence Lemma is a kind of "semantic flattening;" it is done by translating L programs to $\tilde{L}$ programs.

The translation from programs in L to programs in $\tilde{L}$ makes use of a formal semantic definition of L. Although we do not give it here, a definition of L would be presented by defining meaning functions for each of the syntactic classes of L. For instance, the meaning functions for statements, statement lists, and programs would have the following types:

M : $\quad stmt \to state \to state$
M_{sl} : $\quad stmt\_list \to state \to state$
M_p : $\quad program \to state \to state$

Assuming appropriate definitions for M, M_{sl}, and M_p, the language $\tilde{L}$ is defined as follows:

Definition. Programs in $\tilde{L}$ consist only of assignment statements. In addition to scalars and arrays, $\tilde{L}$ incorporates the type *state* (the same type *state* used in the semantic description of L), which associates variables with values. We use "$S := \varnothing$" to denote the initialization of a state variable S with the null state – the state that associates all variables with the value **undefined**; we use

"$S[[v]]$" to denote the value associated with variable v in state S ("the v component of S"); and we use "$S[[v]] := w$" to denote the updating of the v component of state S with w. Arrays are treated like scalars in that they can only be assigned to as a whole. "$S[[a]] := b$" and "a := b" are legal statements in L, while "$S[[a[1]]] := 0$" and "a[1] := 0" are not.

In $\tilde{L}$, an expression on the right-hand side of an assignment statement may contain an application of one of the meaning functions for language L (*i.e.* M, M_{sl}, or M_p) to an appropriate construct of L and to a state.

Example. The following program is a legal program in $\tilde{L}$:

$S := \varnothing$
$S[[y]] := 5$
$z := (M_{sl}[[x := 0;$ **while** $x < 11$ **do** $x := x + y$ **od**$]]S)[[x]]$

Because the x component of $M_{sl}[[x := 0;$ **while** $x < 11$ **do** $x := x + y$ **od**$]]S$ is assigned to variable z, when the program terminates z has the value 15; however, the values of variables x and y are undefined.

We now introduce the Block Equivalence Lemma, which concerns an equivalence property of $\tilde{L}$ programs.

LEMMA. (BLOCK-EQUIVALENCE LEMMA). *Suppose that P and Q are programs in language $\tilde{L}$, that each statement in P occurs in Q and vice versa, and that, except for the set of flow edges whose targets are final-use vertices, $G_P \approx G_Q$. Let V be the set of vertices v such that in both P and Q v is the source of a flow edge whose target is a final-use vertex. Then P and Q are strongly equivalent relative to their imported variables (as the In set) and to the variables defined by the members of V (as the Out set).*

The set V contains those vertices that represent downwards-exposed definitions in *both* program P and program Q. We need the Block-Equivalence Lemma in the form stated above in order to apply it to fragments from a given context; when P and Q are actually program fragments taken from some context, their exported variables are subsets of V.

Example. Consider the following pair of program fragments, which could occur in a context where the exported variables are a and b:

$x := 1$	$x := 2$
$a := x$	$b := x$
$x := 2$	$x := 1$
$b := x$	$a := x$

The two fragments are strongly equivalent relative to the *Out* set $\{a, b\}$, but are not strongly equivalent relative to the *Out* set $\{a, b, x\}$.

PROOF OF THE BLOCK-EQUIVALENCE LEMMA. The variables imported by P and Q consist of all variables for which there is a flow edge in G_P and G_Q out of an initial-definition vertex. Let σ_1 and σ_2 be two states that agree on all the variables imported by P and Q.

Because P and Q contain only assignment statements, there is only a single execution path through each of them. During the execution of each program, we could gather a "trace" of the execution – a sequence of entries that consist of the value assigned at a program statement together with the values of all arguments used to compute that value.

Suppose that P and Q are not strongly equivalent relative to their imported variables (the *In* set) and the variables defined by V (the *Out* set). Then either there is a variable $x \in Out$ that has a different value in the final state for P and the final state for Q, or one of the two programs terminates normally while the other diverges. (An $\tilde{L}$ assignment statement can abort, or can fail to terminate due to a right-hand-side expression like $(M_{sl}[[x := 0;$ **while** $x \geq 0$ **do** $x := x + 1$ **od**$]]S)$).

Case 1. Suppose that programs P and Q both terminate normally, but disagree on the final value of variable x or of component x of the state variable S.

If x is a variable other than the state variable S, then there must be a single flow edge entering the final-use vertex for x (because L doesn't include conditional assignments and because assignments to arrays set the value of the entire array). Call the source of this flow edge vertex v.

If x is a component of the state variable S, then there may be several flow edges entering the final-use vertex for S. However, there is either a unique flow edge whose source represents an assignment to component x, or a unique flow edge whose source represents the initialization of the entire state variable with the null state. Again, call the vertex at the source of the flow edge v.

Variable x (or component x of S) received its final value at v; therefore, there must be at least one variable y (or component y of the state variable) used at v that has a different value in the entry for v in the two traces. This line of reasoning can now be applied to the trace entry for the vertex u for the definition of y that reaches v, and so on. Because each such definition appears at least one entry earlier in the traces, this can continue for no more steps than the length of the trace (*i.e.* the length of programs P and Q). By then we must encounter a vertex where the differing argument variable w has no flow predecessor (*i.e.* the vertex is an initial-definition vertex). The value of such a variable is retrieved from the initial states (σ and σ'). But this leads to a contradiction because w would be one of the imported variables for P and Q and, by assumption, σ and σ' agree on such variables.

Case 2. We must show that it is impossible for one of the two programs, say P, to terminate normally while the

other program, *Q*, does not. The line of reasoning used in Case 1 can be resurrected simply by starting at the vertex *v* at which *Q* diverges, using one of the variables that has a different value in the entry for *v* in the two (partial) traces and is defined at the source of a flow edge that enters *v*. (The argument is non-constructive because it is impossible to decide whether a vertex causes divergence).

We conclude that *P* and *Q* are strongly equivalent relative to their imported variables and those defined by set *V*.

3.4. Semantic Flattening and the Proof of the Equivalence Lemma

The Self-Equivalence Lemma, the definition of language $\tilde{L}$, and the Block Equivalence Lemma were all introduced to help with the StmtList case of the Equivalence Lemma. In this section, we give the proof of the Equivalence Lemma, including the definition of the translation from *L* programs to $\tilde{L}$ programs, which we call "semantic flattening".

PROOF OF THE EQUIVALENCE LEMMA. The proof is by structural induction on the abstract syntax of the programming language. The proof splits into five cases based on the abstract-syntax operator that appears at the root of *T* and *U*. However, four of the five cases, when the operator at the root of *T* and *U* is either an Assign, While, IfThenElse, or Program operator, are demonstrated by (essentially) the argument given in the corresponding case of the Self-Equivalence Lemma. In the proof of the Self-Equivalence Lemma, the convention is that the states σ_i and σ_i' represent sequences of states for two different executions of *T*, one starting in σ_1, the other in σ_1'. To transfer the argument to the Equivalence Lemma one considers the σ_i' sequence to be the sequence for *U*. Because subtrees *T* and *U* correspond, any argument that implies the existence of an edge in *T* also applies to *U*, and *vice versa*.

The one case that does not transfer is Case 4 – when the root operator is the StmtList operator. The argument in Case 4 is an induction over the initial subsequences of *T*; thus, it is the one case of the Self-Equivalence Lemma where it is assumed that the primed and unprimed sequences of states are generated by two executions of the *same* object, namely *T*. The corresponding case of the Equivalence Lemma is different because the components that make up *U* are a *permutation* of the components that make up *T*.

Case 4. The operator at the root of *T* and *U* is the StmtList operator. Let $T_1, T_2, \cdots, T_n$ and $U_1, U_2, \cdots, U_n$ denote the immediate subtrees of *T* and *U*, respectively. Each T_i corresponds to some subtree $U_{\pi(i)}$ that is an immediate subtree of *U*, and *vice versa*, where the mapping $\pi(i)$ is a permutation over the interval $1..n$.

We use *Imp* and *Exp* to denote the imported and exported variables, respectively, of *T* and *U*. We use Imp_i and Exp_i to denote the imported and exported variables, respectively, of T_i and $U_{\pi(i)}$. By the induction hypothesis, T_i is strongly equivalent to $U_{\pi(i)}$ relative to Imp_i and Exp_i.

We need to show that the statement sequences $T_1, T_2, \cdots, T_n$ and $U_1, U_2, \cdots, U_n$ are strongly equivalent relative to *Imp* and *Exp*. To show this, we will translate the two sequences $T_1, T_2, \cdots, T_n$ and $U_1, U_2, \cdots, U_n$ into (straight-line) programs in the language $\tilde{L}$; call the translated sequences $\tilde{T}$ and $\tilde{U}$, respectively. We will show that the translation preserves meaning; we will also show that $\tilde{T}$ and $\tilde{U}$ meet the conditions under which one can apply the Block-Equivalence Lemma. The use of the Block-Equivalence Lemma allows us to overcome the difficulty alluded to earlier, namely that the components that make up *U* are a permutation of the components that make up *T*.

The translation to $\tilde{L}$ is performed as follows: There is a single variable, *S*, of type *state*. For each component, T_i of *T*, we generate three kinds of statements in the order listed below:

(1) The first statement is an assignment statement: $S := \varnothing$.

(2) Then, for each variable $v \in Imp_i$, there is an assignment statement: $S[[[v]]] := v$.

(3) Finally, for each variable $w \in Exp_i$, there is an assignment statement: $w := (M[[T_i]]S)[[[w]]]$.

By the same method, the sequence $U_1, U_2, \cdots, U_n$ is translated to $\tilde{U}$.

The aptness of the translation stems from three properties that we now demonstrate.

Property 1. *T* is strongly equivalent to $\tilde{T}$ relative to *Imp* and *Exp*. (*U* is strongly equivalent to $\tilde{U}$ relative to *Imp* and *Exp*).

Less formally, property 1 can be stated as: The translations of *T* to $\tilde{T}$ and *U* to $\tilde{U}$ preserve the meaning of *T* and *U* (in the context given by the rest of the program).

Proof of Property 1. In the translation of each component T_i of *T*, the state variable *S* is initialized with the current value of every member of Imp_i. Then, assignments are made to the members of Exp_i according to the values they have in the state computed by $M[[T_i]]S$. By the definition of *M*, this is the state that T_i computes on *S*, which, by the Self-Equivalence Lemma agrees on Exp_i with the state computed by T_i from any initial state that agrees with *S* on Imp_i. We conclude that T_i and the translation of T_i are strongly equivalent relative to Imp_i and Exp_i. Consequently, $\tilde{T}$ is a sequence of fragments of straight-line code where each fragment is strongly equivalent to a component T_i of *T* and the fragments are *arranged in the same order* in $\tilde{T}$ as the T_i are in *T*.

The rest of the proof of Property 1 carries over from an argument given in Case 4 of the proof of the Self-Equivalence Lemma; what is necessary to adapt the argument given there is to consider σ_i' to be the state immediately before the sequence of statements in T that represent the translation of T_i. (*Property 1*) □

Property 2. For each variable x in *Exp*, if an assignment $x := (M[\![T_i]\!]S)[[\![x]\!]]$ (from the translation of T_i) reaches the end of T, then the assignment $x := (M[\![U_{\pi(i)}]\!]S)[[\![x]\!]]$ (from the translation of $U_{\pi(i)}$) reaches the end of U.

Our ultimate goal (roughly) is to show that $\tilde{T}$ and $\tilde{U}$ meet the conditions needed to apply the Block Equivalence Lemma. Property 2 asserts that the variables of *Exp* are a subset of the intersection of the downwards-exposed definitions of $\tilde{T}$ and $\tilde{U}$.

Proof of Property 2. If the assignment $x := (M[\![U_{\pi(i)}]\!]S)[[\![x]\!]]$ does not reach the end of U, then there must be an assignment $x := (M[\![U_{\pi(j)}]\!]S)[[\![x]\!]]$ (from the translation of $U_{\pi(j)}$) that reaches the end of U. In this case there would exist a def-order edge, e, in U where the source of e is in $U_{\pi(i)}$ and the target of e is in $U_{\pi(j)}$. Because subtrees T and U correspond, e also occurs in T with the source of e in T_i and the target of e in T_j. However, then the assignment $x := (M[\![T_j]\!]S)[[\![x]\!]]$ would occur after the assignment $x := (M[\![T_i]\!]S)[[\![x]\!]]$, which contradicts the assumption that the latter reaches the end of $\tilde{T}$. (*Property 2*) □

For every statement of the form

$$w := (M[\![T_i]\!]S)[[\![w]\!]]$$

generated by case (3) in the translation of T_i there is a statement

$$w := (M[\![U_{\pi(i)}]\!]S)[[\![w]\!]]$$

generated in the translation of $U_{\pi(i)}$, where in both cases $w \in Exp_i$. In the translations of both T_i and $U_{\pi(i)}$, the assignments to the state S generated by cases (1) and (2) initialize S by the same collection of assignments. (As defined, the initialization statements generated by case (2) may be permuted in the two translations; however, order makes no difference because each initialization statement assigns to a different component of S). By the induction hypothesis, T_i and $U_{\pi(i)}$ are strongly equivalent relative to Imp_i and Exp_i. Consequently, in T we may uniformly substitute $M[\![U_{\pi(i)}]\!]S$ for $M[\![T_i]\!]S$ without altering the meaning of T or any of its flow dependence edges. Call the result of this substitution T'.

The programs $\tilde{T}'$ and $\tilde{U}$ consist of the identical set of statements, although they may be arranged in different orders in the two programs. In order to apply the Block-Equivalence Lemma to $\tilde{T}'$ and $\tilde{U}$, what remains to be shown is that they have the same set of (loop-independent) flow edges.

Property 3. $\tilde{T}'$ and $\tilde{U}$ have the same set of flow edges.

Proof of Property 3. To show that $\tilde{T}'$ and $\tilde{U}$ have the same set of flow edges it is only necessary to show that the flow edges of U are a subset of the flow edges of T'; by demonstrating containment in one direction, the converse holds by symmetry.

Each flow edge in $\tilde{U}$ can be classified as one of three kinds:

(1) An edge that runs from the first statement in the translation of $U_{\pi(i)}$, $S := \varnothing$, to an assignment of the form $x := (M[\![U_{\pi(i)}]\!]S)[[\![x]\!]]$ that is also in the translation of $U_{\pi(i)}$,

(2) An edge that runs from a statement of the form $S[[\![v]\!]] := v$, where $v \in Imp_i$, in the translation of $U_{\pi(i)}$, to an assignment of the form $x := (M[\![U_{\pi(i)}]\!]S)[[\![x]\!]]$ that is also in the translation of $U_{\pi(i)}$.

(3) An edge that runs from a statement of the form $x := (M[\![U_{\pi(i)}]\!]S)[[\![x]\!]]$ in the translation of $U_{\pi(i)}$ (where $x \in Exp_i$), to an assignment of the form $S[[\![x]\!]] := x$ that is in the translation of $U_{\pi(j)}$ (where $x \in Imp_j$).

The edges of types (1) and (2) arise because of the way the translation to L is defined, and thus each edge in U of types (1) and (2) also occurs in T'.

An edge of type (3) occurs when the following conditions hold: (a) $x \in Exp_i$, (b) $x \in Imp_j$, and (c) for all k, such that $\pi(i) < \pi(k) < \pi(j)$, $x \notin Exp_k$ (because translation order follows subtree order, $\pi(i) < \pi(j)$).

The translation of $U_{\pi(j)}$ includes the statement $S[[\![x]\!]] := x$ because $x \in Imp_j$; this can occur because $U_{\pi(j)}$ includes a use of x that is the target of an incoming loop-independent flow edge, or because $U_{\pi(j)}$ includes an assignment to x that is the target of an incoming def-order edge. (The two cases are handled in nearly the same fashion). In either case, because there is no k such that $\pi(i) < \pi(k) < \pi(j)$ for which $x \in Exp_k$, the source of the incoming edge must be in $U_{\pi(i)}$.

T and U have the same edges, so there is a loop-independent flow edge (respectively, def-order edge) from T_i to T_j. All loop-independent flow edges (def-order edges) run left to right, so $i < j$. The only way T' could lack the flow edge from T_i' to T_j' is if there were an intervening assignment to x at T_n', for some n, $i < n < j$. In this case, $x \in Exp_n$, and there would be a def-order edge from T_i to T_n and a loop-independent flow edge (def-order edge) from T_n to T_j. However, there would be corresponding edges in U from $U_{\pi(i)}$ to $U_{\pi(n)}$ and from $U_{\pi(n)}$ to $U_{\pi(j)}$, which contradicts condition (c). We conclude that each edge of type (3) in U occurs in T. (*Property 3*) □

Because $\tilde{T}'$ and $\tilde{U}$ have the identical set of assignment statements, the identical set of flow edges, and, for all variables in *Exp*, the same set of assignments that reach the end of T' and U, T' and U meet the conditions needed to apply the Block-Equivalence Lemma. The Block-Equivalence Lemma implies that $\tilde{T}'$ and $\tilde{U}$ are strongly equivalent relative to *Imp* and *Exp*.

We have now shown (1) that T and $\tilde{T}$ are strongly equivalent relative to *Imp* and *Exp*, (2) that U and $\tilde{U}$ are strongly equivalent relative to *Imp* and *Exp*, and finally (3) that T (really T') and U are strongly equivalent relative to *Imp* and *Exp*. Thus, we conclude that T and U are strongly equivalent relative to *Imp* and *Exp*. □

3.5. Proof of the Equivalence Theorem

The Equivalence Theorem follows as a corollary of the Equivalence Lemma.

THEOREM. (EQUIVALENCE THEOREM). *If P and Q are programs for which $G_P \approx G_Q$, then P and Q are strongly equivalent.*

PROOF. By the Equivalence Lemma, P and Q are strongly equivalent relative to their imported variables (as the *In* set) and their exported variables (as the *Out* set). By the relativized strong equivalence of P and Q, we know that for an arbitrary initial state σ, either P and Q both diverge or they produce final states σ_P and σ_Q, respectively, that agree on their exported variables. However, the exported variables of P and Q consist of all variables that are assigned to in the two programs, so for all variables that are not in the exported set σ_P and σ_Q must agree with σ, and hence with each other. Consequently, P and Q are strongly equivalent. □

4. Minimality of Program Dependence Graphs

In choosing which data-dependency edges to include in our definition of program dependence graphs we had two goals: (1) The program dependence graphs of inequivalent programs should not be isomorphic; (2) Program dependence graphs should be minimal in the sense that omitting any class of data-dependency edges permits inequivalent programs to have isomorphic program dependence graphs. Section 3 showed that our definition of program dependence graphs meets goal (1); in this section we illustrate how each class of data-dependency edges included in our definition of program dependence graphs is needed to meet goal (2), by demonstrating some sample inequivalent programs that would be indistinguishable if program dependence graphs were to lack a particular class of edge.

The distinction between loop-independent and loop-carried flow dependencies is necessary to distinguish between the following two program fragments:

```
x := 0                          x := 0
while P do                      while P do
   y := x                          if Q then x := 1 fi
   if Q then x := 1 fi             y := x
od                              od
```

The program dependence graphs for these fragments have identical vertices, control dependency edges, and def-order dependency edges. If we ignore the distinction between loop-independent and loop-carried flow dependencies, they have identical flow dependency edges as well; however, in the left-hand fragment, the flow dependency from the assignment statement $x := 1$ to the assignment $y := x$ is a loop-carried dependency, whereas the corresponding dependency in the right-hand fragment is a loop-independent one.

Def-order dependencies are needed in program dependence graphs to be able to distinguish between the program fragments:

```
if P then x := 0 fi             if Q then x := 1 fi
if Q then x := 1 fi             if P then x := 0 fi
y := x                          y := x
```

Here the program dependence graphs for these fragments have identical vertices, control dependency edges, and flow dependency edges. If program dependence graphs did not contain def-order dependency edges, these programs would have identical program dependence graphs, although they do not have equivalent behaviors. Including def-order dependences causes them to have different program dependence graphs; in the left-hand fragment, there is a def-order dependency from the assignment statement $x := 0$ to the assignment $x := 1$, whereas in the right-hand fragment, the def-order dependency runs in the other direction, from $x := 1$ to $x := 0$.

5. The Equivalence Theorem and Standard Data Dependencies

The data dependencies used in this paper are somewhat non-standard. Ordinarily, two kinds of data dependencies, called *anti-dependencies* and *output dependencies* are used in addition to flow dependencies.[3] We omit anti-dependencies and replace output dependencies with *def-order* dependencies, first introduced in [7]. In this section we discuss the differences between def-order and output dependencies including one potential advantage of the former, and argue that, because def-order dependencies are more general than output dependencies, the Equivalence Theorem holds under either definition of program dependence graphs.

Although def-order dependencies resemble output dependencies in that they both relate two assignments to

[3] As with flow dependencies, anti-dependencies and output dependencies may be further characterized as loop independent or loop carried.

the same variable, they are two different concepts. An output dependency $v_1 \longrightarrow_o v_2$ between two definitions of x can hold only if there is no intervening definition of x along some execution path from v_1 to v_2; however, there can be a def-order dependency $v_1 \longrightarrow_{do} v_2$ between two definitions even if there is an intervening definition of x along *all* execution paths from v_1 to v_2. This situation is illustrated by the following example program fragment, which demonstrates that it is possible to have a program in which there is a dependency $v_1 \longrightarrow_{do} v_2$ but not $v_1 \longrightarrow_o v_2$, and *vice versa*:

```
[1]     x := 10
[2]     if P then
[3]             x := 11
[4]             x := 12
[5]     fi
[6]     y := x
```

The one def-order dependency, $[1] \longrightarrow_{do([6])} [4]$, exists because the assignments to x in lines [1] and [4] both reach the use of x in line [6]. In contrast, the output dependencies are $[1] \longrightarrow_o [3]$ and $[3] \longrightarrow_o [4]$, but there is no output dependency $[1] \longrightarrow_o [4]$.

Ideally, program dependence graphs should be defined so that equivalent programs whose abstract syntax trees differ only in the orderings of statements have isomorphic program dependence graphs. The following programs provide an example for which def-order dependencies achieve this goal, while output dependencies do not:

```
x := 0;          x := 1;
a := x;          b := x;
x := 1;          x := 0;
b := x;          a := x;
x := 2           x := 2
```

These two programs are strongly equivalent; their program dependence graphs have the same (empty) set of def-order dependencies, but have different sets of output dependencies.

Although def-order dependencies may be preferable to output dependencies, past work has used the latter; thus, it is important to know that the Equivalence Theorem still holds if the program dependence graph is defined to have output dependency edges rather than def-order dependency edges. A program's def-order dependency edges can be determined given the flow dependency edges and loop-independent output dependency edges. Therefore, if two programs have isomorphic program dependence graphs defined using output dependencies, they also have isomorphic program dependence graphs defined using def-order dependencies. Consequently, by the Equivalence Theorem, they are strongly equivalent.

References

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

2. Allen, J.R. and Kennedy, K., "PFC: A program to convert Fortran to parallel form," Technical Report MASC TR82-6, Department of Math. Sciences, Rice University, Houston, TX (March 1982).

3. Allen, J.R., "Dependence analysis for subscripted variables and its application to program transformations," Ph.D. Thesis, Department of Math. Sciences, Rice University, Houston, TX (April 1983).

4. Bannerjee, U., "Speedup of ordinary programs," Ph.D. Thesis, University of Illinois, Urbana, IL (October 1979).

5. 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).

6. Horwitz, S., Prins, J., and Reps, T., "Integrating non-interfering versions of programs," Technical Report 690, Department of Computer Sciences, University of Wisconsin—Madison (March 1987).

7. Horwitz, S., Prins, J., and Reps, T., "Integrating non-interfering versions of programs," pp. 133-145 in *Conference Record of the Fifteenth ACM Symposium on Principles of Programming Languages*, (San Diego, CA, January 13-15, 1988), (1988).

8. 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 Transactions on Computers C-21*, pp. 1293-1310 (December 1972).

9. Kuck, D.J., *The Structure of Computers and Computations, Vol. 1*, John Wiley and Sons, New York, NY (1978).

10. 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), (1981).

11. Ottenstein, K.J. and Ottenstein, L.M., "The program dependence graph in a software development environment," *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, (Pittsburgh, PA, April 23-25, 1984), ACM SIGPLAN Notices* **19**(5) pp. 177-184 (May 1984).

12. Towle, R., "Control and data dependence for program transformations," TR 76-788, Department of Computer Science, University of Illinois, Urbana-Champaign, IL (March 1976).

13. Wolfe, M. J., "," Rep. 82-1105, University of Illinois, Urbana, IL (October 1982).

Conference Record of the 15th Annual ACM Symposium on Principles of Programming Languages