CORRECTNESS OF AN ALGORITHM FOR RECONSTITUTING A PROGRAM FROM A DEPENDENCE GRAPH

by

Thomas Ball, Susan Horwitz & Thomas Reps

Computer Sciences Technical Report #947

July 1990

Correctness of an Algorithm for Reconstituting a Program from a Dependence Graph

THOMAS BALL, SUSAN HORWITZ and THOMAS REPS
University of Wisconsin – Madison

Given an arbitrary program dependence graph, the algorithm ReconstituteProgram determines whether the graph is *feasible* (*i.e.*, is the dependence graph of some program), and if so produces such a program. ReconstituteProgram is currently used for program integration, and has the potential to be used by other algorithms that manipulate program dependence graphs. This paper corrects a minor error that was in the original definition of ReconstituteProgram, and proves that the new version of the algorithm is correct.

1. INTRODUCTION

A program dependence graph G is *feasible* if there exists some program P such that G is P's program dependence graph. Given an arbitrary program dependence graph G, the algorithm ReconstituteProgram determines whether G is feasible and, if G is feasible, produces a program corresponding to G.

ReconstituteProgram is vital to the program-integration algorithm of [Horwitz89]. It also has potential for use by optimizing, vectorizing, or parallelizing compilers that perform transformations on dependence graph representations of programs. Given the key role of this algorithm, we considered it important to provide a proof of its correctness. In fact, while attempting to prove the correctness of the algorithm defined in [Horwitz89], we discovered an error. This paper provides a correct version of ReconstituteProgram (Section 3) as well as a proof of its correctness (Section 4).

There have been a number of dependence graph representations defined in the literature; the definition under consideration here is given in Section 2. An important characteristic of these dependence graphs is that their control-dependence edges impose a *tree* structure on the graph, albeit a tree in which a vertex's children are unordered. The crux of the program-reconstitution problem is to define a total order for each vertex's control-dependence children; given these total orders, the control dependence subgraph of G can be easily converted to an abstract syntax tree and then to a program. Given a feasible graph G, a correct program-reconstitution algorithm defines a total order for the children of each vertex in the tree such that the corresponding program's dependence graph is G. Such a set of total orders (*i.e.*, one total order for the children of each vertex) is called a *good total order* for graph G.

By definition, a feasible graph has one or more good total orders. The proof of correctness for ReconstituteProgram has two main parts. The first part of the proof (Section 4.1) shows that given a feasible graph G, the total order defined by ReconstituteProgram for the children of an arbitrary individual vertex of G respects at least one good total order of G (*i.e.*, $\forall$ vertices $v \in G$, $\exists$ good total order t, such that ReconstituteProgram's order for the children of v respects t). The second part of the proof (Section 4.2) shows that the union of the total orders defined for each individual vertex's children is a good total order

This work was supported in part by a David and Lucile Packard Fellowship for Science and Engineering, by the National Science Foundation under grants DCR-8552602 and CCR-8958530, 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, Xerox, and Kodak.

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

(*i.e.*, ∃ good total order t, such that ∀ vertices v ∈ G, ReconstituteProgram's order for the children of v respects t).

2. PROGRAM DEPENDENCE GRAPHS

NOTATION. The following notation is used throughout the paper: capital letters (G, R, P) represent graphs, subgraphs and programs; bold lowercase letters (**v**, **w**, **r**) represent the vertices of a graph; italic lowercase letters (x, y) represent variables.

We assume that programs are written in a simple language in which expressions contain scalar variables and constants, and the only statements are assignment statements, conditional statements, while loops, and end statements. An end statement, which can only appear at the end of the program, names all the variables that are of interest as output from the program; by definition, only the variables named in the end statement have values in the final state. Although the language does not include read statements, variables can be used before being defined; these variables' values come from the initial state.

The program dependence graph under consideration here is as defined in [Horwitz89]. The program dependence graph for 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:

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

(2) For each variable x for which there is a path in the standard control-flow graph for P on which x is used before being defined (see [Aho86]), there is a vertex called the *initial definition of* x. This vertex represents an assignment to x from the initial state. The vertex is labeled "x := InitialState(x)."

(3) For each variable x named in P's end statement, there is a vertex called the *final use of* x. It represents an access to the final value of x computed by P, and is labeled "FinalUse(x)".

The edges of G_P represent *dependences* among program components. An edge represents either a *control dependence* or a *data dependence*. Control dependence edges are labeled either **true** or **false**, and the source of a control dependence edge is always the entry vertex or a predicate vertex. A control dependence edge from vertex **v** to vertex **w**, denoted by $\mathbf{v} \longrightarrow_c \mathbf{w}$, means that during execution, whenever the predicate represented by **v** is evaluated and its value matches the label on the edge to **w**, then the program component represented by **w** will eventually be executed if the program terminates. A method for determining control dependence edges for arbitrary programs is given in [Ferrante87]; however, because we are assuming that programs include only assignment, conditional, and while statements, the control dependence edges of G_P can be determined in a much simpler fashion. For the language under consideration here, the control dependences reflect a program's nesting structure; program dependence graph G_P contains a *control dependence edge* from vertex **v** to vertex **w** of G_P iff one of the following holds:

(1) **v** is the entry vertex, and **w** represents a component of P that is not nested within any loop or conditional; these edges are labeled **true**.

(2) **v** represents a control predicate, and **w** represents a component of P immediately nested within the loop or conditional whose predicate is represented by **v**. If **v** is the predicate of a while-loop, the

---

[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.

edge $v \rightarrow_c w$ is labeled **true**; if v is the predicate of a conditional statement, the edge $v \rightarrow_c w$ is labeled **true** or **false** according to whether w occurs in the **then** branch or the **else** branch, respectively.[2]

A data dependence edge from vertex v to vertex w means that the program's computation might be changed if the relative order of the components represented by v and w were reversed. In this paper, program dependence graphs contain two kinds of data-dependence edges, representing *flow dependences* and *def-order dependences*. The data-dependence edges of a program dependence graph can be computed using standard data-flow analysis techniques.

A program dependence graph contains a flow dependence edge from vertex v to vertex w iff all of the following hold:

(1) v is a vertex that defines variable x.

(2) w is a vertex that uses x.

(3) Control can reach w after v 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 by which the definition of x at v reaches the use of x at w. (Initial definitions of variables are considered to occur at the beginning of the control-flow graph; final uses of variables are considered to occur at the end of the control-flow graph.)

A flow dependence that exists from vertex v to vertex w, where v and w define variable x, is denoted by $v \rightarrow_f^x w$. When the variable defined by v and w is irrelevant, the notation $v \rightarrow_f w$ is used.

Flow dependences can be further classified as *loop carried* or *loop independent*. A flow dependence $v \rightarrow_f w$ is carried by loop L, denoted by $v \rightarrow_{lc(L)} w$, if in addition to (1), (2), and (3) above, the following also hold:

(4) There is an execution path that both satisfies the conditions of (3) above and includes a backedge to the predicate of loop L.

(5) Both v and w are enclosed in loop L.

A flow dependence $v \rightarrow_f w$ is loop independent, denoted by $v \rightarrow_{li} w$, if in addition to (1), (2), and (3) above, there is an execution path that satisfies (3) above and includes *no* backedge to the predicate of a loop that encloses both v and w. It is possible to have both $v \rightarrow_{lc(L)} w$ and $v \rightarrow_{li} w$.

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

(1) v and w both define the same variable.

(2) v and w are in the same branch of any conditional statement that encloses both of them.

(3) There exists a program component u such that $v \rightarrow_f u$ and $w \rightarrow_f u$.

(4) v occurs to the left of w in the program's abstract syntax tree.

A def-order dependence from v to w with "witness" u is denoted by $v \rightarrow_{do(u)} w$.

---

[2] In other definitions that have been given for control dependence edges, there is an additional edge from each predicate of a while statement to itself labeled **true**. This kind of edge is left out of our definition because it is not necessary for our purposes.

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 dependence edge between two vertices, each is labeled by a different loop that carries the dependence. When there is more than one def-order edge between two vertices, each is labeled by a different "witness" vertex.

3. RECONSTITUTING A PROGRAM FROM AN ARBITRARY PROGRAM DEPENDENCE GRAPH

Given an arbitrary program dependence graph G, function ReconstituteProgram must determine whether G is feasible (*i.e.*, corresponds to some program), and if it is, create an appropriate program from G.

Because we are assuming a restricted set of control constructs, each vertex of G has at most one incoming control dependence edge (from a predicate vertex or the entry vertex); *i.e.*, the control dependences of G define a tree rooted at the entry vertex. The crux of the program-reconstitution problem is to determine, for each predicate vertex v (and for the entry vertex as well), an ordering on the targets of v's outgoing control dependence edges that is consistent with the data dependences of G. Once all vertices are ordered, the control dependence subgraph of G can be easily converted to an abstract-syntax tree.

The rest of this section describes the function ReconstituteProgram. Most of this description is taken from [Horwitz89]; however, the original definition of ReconstituteProgram given there contained a minor error, which has been corrected here (see Section 3.3 and Figure 7).

ReconstituteProgram is presented in outline form in Figure 1. ReconstituteProgram alters graph G_C, which is a copy of G; G itself is saved, unaltered, for use in the test on line [9]. In the for-loop (lines [2]-[7]), the tree induced on G_C by its control dependences is traversed in post-order. For each vertex v visited during the traversal, an attempt is made to determine an acceptable order for v's children; this attempt is performed by the procedure OrderRegion, which is explained in detail below. We assume that a function, named TransformToSyntaxTree, has been provided to convert G_C with ordered vertices into the corresponding abstract-syntax tree.

ReconstituteProgram can fail in two different ways. Failure can occur because procedure OrderRegion determines that there is no acceptable ordering for the children of some vertex. Failure can also occur at a later point, after OrderRegion succeeds in ordering all vertices of G_C. If OrderRegion succeeds, TransformToSyntaxTree is used to produce program P from G_C, P's program dependence graph G_P is built, and G_P is compared to G; failure occurs if G and G_P are not identical.

3.1. Procedure OrderRegion: Ordering vertices within a region

DEFINITION. The subgraph induced on a collection of vertices, all of which are targets of control dependence edges from some vertex v, is called a *region*; v is the *region head*. If v represents the predicate of a conditional, v is the head of *two* regions; one region includes all statements in the "true" branch of the conditional, the other region includes all statements in the "false" branch of the conditional. For all vertices w, EnclosingRegion(w) is the region that includes w (*not* the region of which w is the head).

NOTATION. R is used to denote a region; Both r and head(R) are used to denote the vertex that is the head of region R.

Given region R, the main job of procedure OrderRegion (shown in Figure 2) is to find a total ordering of the vertices of R that preserves the flow and def-order dependences of G_C, or to discover that no such ordering is possible. Note that simply using a topological ordering of the region is not satisfactory. For example, consider the dependence graph fragment shown in Figure 3. A topological ordering of the vertices of region headed by vertex c is: f, d, g, e; however, the dependence graph of the program generated

```
function ReconstituteProgram(G) returns a program or FAILURE
declare
   G, G_P: program dependence graphs
   G_C: a graph
   v, w: vertices of G_C
begin
[1]  G_C := a copy of G
[2]  for each vertex v of G_C in a post-order traversal of the control-dependence subgraph of G_C do
[3]     if OrderRegion(G_C, { w | (v --T_c--> w) ∈ E(G_C) }) fails then return( FAILURE ) fi
[4]     if v represents an if-predicate then
[5]        if OrderRegion(G_C, { w | (v --F_c--> w) ∈ E(G_C) }) fails then return( FAILURE ) fi
[6]     fi
[7]  end
[8]  P := TransformToSyntaxTree(G_C);
[9]  if G = G_P then return( P )
[10] else return( FAILURE )
[11] fi
end
```

Figure 1. The operation ReconstituteProgram(G) creates a program corresponding to the program dependence graph G by ordering all vertices, or discovers that G is infeasible.

```
procedure OrderRegion(G_C,R)
declare
   G_C: a graph
   R: a region of G_C
begin
   PreserveExposedUsesAndDefs(G_C,R)
   if PreserveSpans(R) fails then fail else TopSort(R) fi
   ProjectUsesAndDefs(G_C,R)
end
```

Figure 2. Procedure OrderRegion adds new edges to the given region to ensure that dependences are respected, topologically sorts the vertices of the region, and projects information onto the region head.

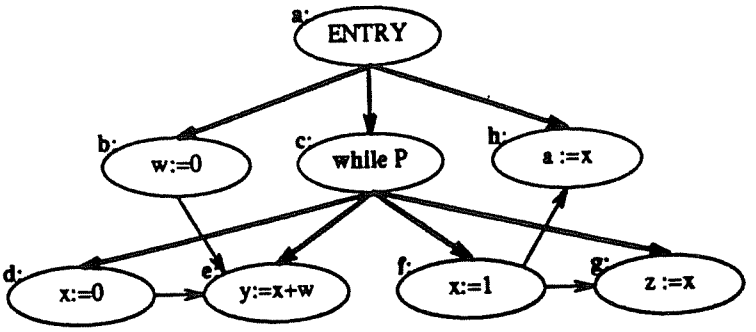

Figure 3. Dependence graph fragment: Topological ordering f, d, e, g, of the vertices of the region headed by vertex c is not acceptable.

according to this ordering would incorrectly have flow edges from d to g and from d to h, rather than the ones from f to g and from f to h.

A secondary responsibility of OrderRegion is to project onto the head of R information from the vertices of R regarding variable uses, variable definitions, and incoming and outgoing edges. This projection ensures that, when the head of R is considered as a vertex in its enclosing region, it represents all uses and definitions that occur in R.

To order the vertices of R, OrderRegion calls procedures PreserveExposedUsesAndDefs and PreserveSpans (discussed below). These procedures add edges to R to force an ordering of the vertices consistent with the region's data dependences. (This process is roughly that of introducing anti- and output dependences consistent with the flow and def-order dependences of region R.) If this process introduces a cycle in R, OrderRegion fails; otherwise, a topological sort of region R produces an ordering consistent with the region's data dependences.

Information is projected onto the head of region R both by procedure PreserveExposedUsesAndDefs, which projects the loop-carried flow edges of R and the edges of G_C with only a single endpoint in R onto the region head, and by procedure ProjectUsesAndDefs, which projects onto the head of R information from the vertices in region R about variable uses and definitions. For example, procedure ProjectUsesAndDefs would designate vertex c of Figure 3 as representing uses of w and x, and definitions of x, y, and z. In other words, a vertex v represents a use (definition) of variable p if v contains a use of (or assignment to) p, or if a control-descendant of v contains a use of (or assignment to) p.

3.2. Procedure PreserveExposedUsesAndDefs: Preserving upwards-exposed uses and downwards-exposed definitions

For all variables x, a use of x that is upwards-exposed [Aho86] within a region must precede all definitions of x within the region other than its loop-independent flow-predecessors (a use of x can be upwards-exposed and still have a loop-independent flow-predecessor that defines x within the region if the flow-predecessor represents a conditional definition). Vertex e in Figure 3 represents an upwards-exposed use of variable w.

Similarly, a definition of x that is downwards-exposed within a region must follow all other definitions of x within the region other than those to which it has a def-order edge (again, a definition of x can be downwards-exposed and still precede a conditional definition of x). Vertex **f** in the example of Figure 3 represents a downwards-exposed definition of variable x.

Procedure PreserveExposedUsesAndDefs uses flow edges of G_C having only one endpoint inside the given region R, and loop-carried flow edges having both endpoints inside R to identify exposed uses and definitions. It then adds edges to R to ensure that exposed uses and definitions are ordered correctly with respect to other definitions within the region. Finally, the edges used to identify exposed uses and definitions are removed from R and are projected onto the region head. Def-order edges with a single endpoint inside R are also projected onto head(R). This ensures that the region that includes the head of R will be ordered correctly during a future call to OrderRegion. PreserveExposedUsesAndDefs performs the following four steps:

Step (1): Identify upwards-exposed uses.

A vertex with an incoming loop-independent flow edge whose source is outside region R, or with an incoming loop-carried flow edge with arbitrary source, represents an *upwards-exposed use* of the variable x defined at the source of the flow edge. Mark each such vertex UPWARDS-EXPOSED-USE(x).

Step (2): Identify downwards-exposed definitions.

A vertex that represents a definition of variable x and has an outgoing loop-independent flow edge whose target is outside region R, or has an outgoing loop-carried flow edge with arbitrary target, represents a downwards-exposed definition of x.[3] Mark each such vertex DOWNWARDS-EXPOSED-DEF(x).

Step (3): Preserve exposed uses and definitions.

For each vertex **w** marked UPWARDS-EXPOSED-USE(x), add a new edge from **w** to all vertices **v** in the region such that **v** represents a definition of variable x, and **v** is not a loop-independent flow predecessor of **w**. For each vertex **w** marked DOWNWARDS-EXPOSED-DEF(x), add a new edge to **w** from all vertices **v** in the region such that **v** represents a definition of x and there is no def-order edge from **w** to **v**.

Step (4): Project edges onto the region head.

Let S stand for $R \cup \{ \text{head}(R) \}$. Replace all flow and def-order edges with source outside of S and target inside S with an edge (of the same kind) from the source to head(R). Replace all flow and def-order edges with source inside S and target outside of S with an edge (of the same kind) from head(R) to the target.

Consider each loop-carried flow edge $\mathbf{v} \longrightarrow_{lc(L)} \mathbf{w}$ such that both **v** and **w** are in S. If head(R) = L, then remove the edge; otherwise, replace the edge with a loop-carried flow edge $\text{head}(R) \longrightarrow_{lc(L)} \text{head}(R)$.

Figure 4 shows the example dependence graph fragment of Figure 3 after the four steps described above have been performed on the region headed by vertex **c**. The edge from **d** to **f** was added in Step (3), due to **f** being downwards-exposed, and prevents **f** from preceding **d** in a topological ordering. The edges from **b**

[3]Our use of the term "downwards-exposed" is slightly nonstandard; we consider a definition to be downwards-exposed in code segment C only if it reaches the end of C and the variable it defines is live at the end of C.

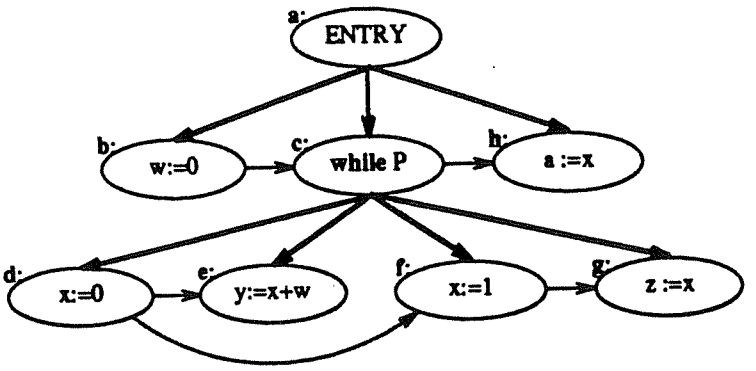

Figure 4. Dependence graph fragment with new edge **d**→**f** added to preserve the downwards-exposed definition of x at vertex **f**.

to **c** and from **c** to **h** were added in Step (4), replacing those from **b** to **e** and **f** to **h**, respectively.

3.3. Dependences induced by spans

To simplify this section's presentation, we begin by considering regions that only include assignment statements; under this restriction, in a feasible PDG, each use of variable x within a region is reached by at most one definition of x that occurs within the region.

In the example dependence graph fragment of Figure 4, the ordering **d**, **f**, **e**, **g** of the vertices subordinate to vertex **c** is a topological ordering, but an unacceptable one for our purposes. The problem with this ordering is that it allows the definition of variable x at vertex **f** to "capture" the use of x at vertex **e**. The dependence graph of the program generated according to this ordering would incorrectly have a flow edge from **f** to **e**, rather than the one from **d** to **e**. In general, a definition **d** of variable x must precede all uses it reaches via loop-independent flow edges; other definitions of x must either precede **d** or follow all the uses reached by **d**. This observation leads to the following definition:

DEFINITION. The *span* of a definition **d**, where **d** defines variable x, is the set {**d**}, together with all uses of x that are loop-independent flow targets of **d** and in the same region as **d**.

$$\text{Span}(\mathbf{d}, x) = \{\, \mathbf{d} \,\} \cup \{\, \mathbf{u} \mid (\mathbf{d} \longrightarrow^{x}_{li} \mathbf{u}) \in E(\text{EnclosingRegion}(\mathbf{d})) \,\}$$

Span(**d**, x) is called an x-span, and vertex **d** is its *head*.

Restating the observation above in terms of spans, a definition $\mathbf{d}_1$ of variable x must precede all vertices in Span($\mathbf{d}_1, x$); other definitions of x must either precede $\mathbf{d}_1$ or follow all vertices in Span($\mathbf{d}_1, x$). Furthermore, for any other x-span with head $\mathbf{d}_2$, if *any* vertex in Span($\mathbf{d}_1, x$) must precede a vertex in Span($\mathbf{d}_2, x$), then *all* vertices in Span($\mathbf{d}_1, x$) must precede $\mathbf{d}_2$.

Unacceptable topological orderings are excluded by considering, for each variable x, all pairs <$\mathbf{d}_1$, $\mathbf{d}_2$> of definitions of x. If there is some vertex **v** in Span($\mathbf{d}_1, x$) that must precede some vertex **w** in Span($\mathbf{d}_2, x$), (because of a path from **v** to **w**) then edges are added from all vertices in Span($\mathbf{d}_1, x$) – Span($\mathbf{d}_2, x$) to

vertex d_2. Similarly, if there is a path from a vertex in Span(d_2, x) to a vertex in Span(d_1, x), edges are added from all vertices in Span(d_2, x) – Span(d_1, x) to vertex d_1. For example, in the graph fragment of Figure 4, the edge $e \rightarrow f$ would be added because the edge $d \rightarrow f$ (introduced by PreserveExposedUsesAndDefs) forms a path from Span(d, x) to Span(f, x), and vertex e is in Span(d, x) – Span(f, x).

The reason for taking the set difference Span(d_1, x) – Span(d_2, x), is that even in regions containing only assignment statements, spans can overlap, as illustrated in Figure 5. Because c is itself in Span(b, x), adding edges from *all* vertices in Span(b, x) to c would create a self-loop at c, making a topological ordering impossible.

Allowing vertices that represent loops and conditionals introduces the possibility that spans may overlap in two new ways, as illustrated in Figure 6. The first case in Figure 6 does not require any special handling; since there is a path from d_1 to d_2, the technique described above will add edges from all vertices in Span(d_1, x) – Span(d_2, x) to vertex d_2, and the two spans will be ordered correctly.

The second case in Figure 6 does require a modification to the technique described above. Note that if G is feasible, there must be a def-order dependence edge from d_1 to d_2 or *vice versa*; without loss of generality, assume $d_1 \rightarrow_{do} d_2$. Since there is a path from a vertex in Span(d_2, x) to a vertex in Span(d_1, x), (namely, the edge from d_2 to u), the technique described above would add edges from all vertices in Span(d_2, x) – Span(d_1, x) to vertex d_1. This would include adding an edge from d_2 itself to d_1, thus creating a cycle (because of the def-order edge from d_1 to d_2). The required modification to the technique described above is to look for a path from a vertex in Span(d_1, x) to a vertex in Span(d_2, x) only if there is no def-order edge from d_1 to d_2. (The algorithm published in [Horwitz89] erroneously omits this modification.)

There may be pairs of spans, Span(d_1, x) and Span(d_2, x), such that there is no path in either direction between Span(d_1, x) and Span(d_2, x); such pairs are called *independent x-span pairs*. It is still necessary to add edges to force one span to precede the other so as to exclude unacceptable topological orderings. Although it might seem that an arbitrary choice can be made, there are examples in which making the wrong choice leads to the introduction of a cycle in a fragment of a feasible graph.

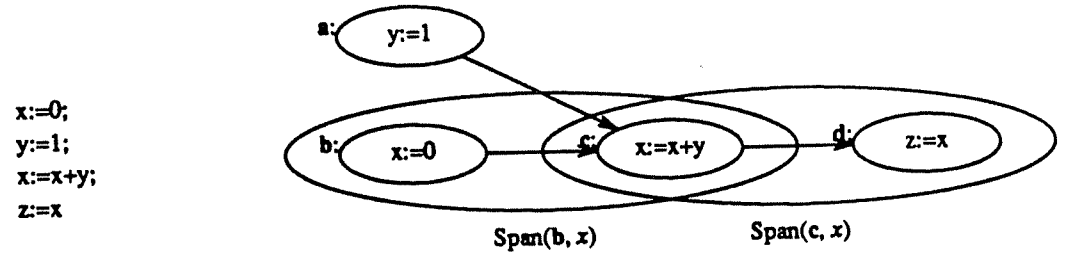

Figure 5. Straight-line code fragment and corresponding dependence graph fragment (control edges omitted) with overlapping x-spans.

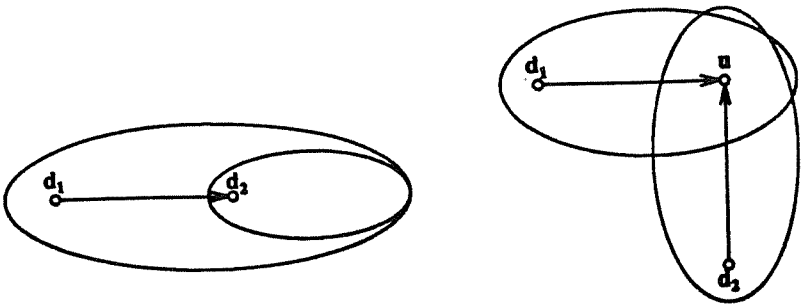

Figure 6. Conditionals and loops can lead to the two additional kinds of overlapping spans shown above. Vertices d_1 and d_2 represent definitions of variable x; vertex u represents a use of variable x.

Unfortunately, the problem of determining the right choice in such situations is NP-complete [Horwitz87]. However, we expect that in practice there will be very few such choices to be made, and a simple backtracking algorithm will suffice: if a cycle is introduced when ordering spans, procedure PreserveSpans backtracks to the most recent choice point, and tries a different choice. If all choices lead to the introduction of a cycle, the graph is infeasible.

3.4. Procedure PreserveSpans

Procedure PreserveSpans is presented in Figure 7. PreserveSpans makes use of an auxiliary procedure, OrderDependentSpans, to order any span pairs of region R whose relative order is forced by a connecting path. An invariant of the two procedures, established in the first line of PreserveSpans, is that graph R is transitively closed. The basic operation used in PreserveSpans and OrderDependentSpans is "AddEdgeAndClose(R, **a**→**b**)", whose first argument is a graph and whose second argument is an edge to be added to the graph. AddEdgeAndClose(R, **a**→**b**) carries out two actions:

(1) Edge **a**→**b** is inserted into R.

(2) Any additional edges needed to transitively close R are inserted into R.

Because R is transitively closed, paths that force span orderings correspond to edges of R; furthermore, the cost of AddEdgeAndClose is quadratic (rather than cubic) in the number of vertices of R.

Each edge of R can be *marked* or *unmarked*; the edges added to R by AddEdgeAndClose (by either (1) or (2)) are unmarked. Edges are marked at line [1] in OrderDependentSpans. An invariant of the while loop in OrderDependentSpans is that for each marked edge e, all spans for which e forces an ordering are appropriately ordered. Thus, after an unmarked edge **v**→**w** is selected (and marked), the invariant is reestablished as follows: line [2] generates all variables x for which both **v** and **w** are elements of an x-span (but not necessarily the same x-span); lines [3] and [4] iterate over all pairs of x-spans (represented by their heads) such that **v** is a member of the first span and **w** is a member of the second; line [5] orders the two spans as forced by the presence of edge **v**→**w**.

```
procedure PreserveSpans(R)
declare
   R: a region
   h1, h2: vertices of R
   Stack: a stack
begin
   TransitivelyClose(R)
   if R is cyclic then fail fi
   Unmark all edges of R
   OrderDependentSpans(R)
   Stack := EmptyStack()
   do
      R is acyclic and there exist independent x-span pairs (for some variable x) with heads h1 and h2 →
         Push(Stack, R, h1, h2)
         AddEdgeAndClose(R, h1→h2)
         OrderDependentSpans(R)
    [] R is cyclic and Empty(Stack) → fail
    [] R is cyclic and ¬Empty(Stack) →
         R, h1, h2 := Pop(Stack)
         AddEdgeAndClose(R, h2→h1)
         OrderDependentSpans(R)
   od
end

procedure OrderDependentSpans(R)
declare
   R: a region
   a, b, c, u, v, w: vertices of R
   A, B: sets of vertices
   x: a variable
begin
   while there exists an unmarked edge v→w in R do
[1]   Mark edge v→w
[2]   for each variable x ∈ (Defs(v) ∪ Uses(v)) ∩ (Defs(w) ∪ Uses(w)) do
         /* v is in an x-span and w is in an x-span */
         A := { u | v ∈ Span(u, x) }  /* heads of x-spans of which v is a member */
         B := { u | w ∈ Span(u, x) }  /* heads of x-spans of which w is a member */
[3]      for each vertex a ∈ A do
[4]         for each vertex b ∈ B do
[5]            if b →do a ∉ E(R) then
                  for each c ∈ ( Span(a, x) − Span(b, x) ) do
                        if c→b ∉ E(R) then AddEdgeAndClose(R, c→b) fi
                  end
               fi
            end
         end
      end
   end
end
```

Figure 7. Procedure PreserveSpans introduces edges into region R to preserve the spans of R. The test at line [5] was erroneously omitted from this procedure in [Horwitz89].

The initial call on OrderDependentSpans in PreserveSpans serves to introduce edges for all forced span orderings. The **do-od** loop then implements a backtracking algorithm that examines all choices for independent span pairs. Each pair of independent spans (represented by their span heads, say h_1 and h_2)

represents two possibilities—the elements of Span($\mathbf{h}_1$, x) could precede the elements of Span($\mathbf{h}_2$, x), or *vice versa*. The first possibility is represented by the call AddEdgeAndClose(R, $\mathbf{h}_1 \rightarrow \mathbf{h}_2$), which introduces an edge directed from $\mathbf{h}_1$ to $\mathbf{h}_2$; the second possibility (which is tried only in the backtracking step, guarded by the condition "R is cyclic and ¬Empty(*Stack*)") is represented by the call AddEdgeAndClose(R, $\mathbf{h}_2 \rightarrow \mathbf{h}_1$). In both cases, OrderDependentSpans is called to introduce edges for all span orderings forced as a consequence of the new edge. (A single edge, such as $\mathbf{h}_1 \rightarrow \mathbf{h}_2$, may force an ordering between spans other than those headed by $\mathbf{h}_1$ and $\mathbf{h}_2$.)

The information needed for backtracking is kept as a stack of triples: the graph R as it existed before a given "choice" (including the saved marks on R's edges), span head $\mathbf{h}_1$, and span head $\mathbf{h}_2$. Backtracking terminates with failure if R is cyclic and the stack is empty, because no alternative remains to be tried. When R is cyclic but the stack is not empty, one entry is popped from the stack and the "choice" is tried in the opposite direction. (Since there are only two choices to be tried for each pair of span heads, there is no Push before continuing the search with the second alternative.) PreserveSpans terminates with success if R is acyclic and there remain no independent x-span pairs.

4. PROOF OF CORRECTNESS OF RECONSTITUTE PROGRAM

THEOREM (Correctness of ReconstituteProgram). *ReconstituteProgram(G) succeeds iff graph G is feasible.*

PROOF:

(1) **ReconstituteProgram succeeds ⇒ G is feasible.** When ReconstituteProgram(G) succeeds, it returns a program P such that $G = G_P$ (by the test in line [9] of Figure 1). Therefore, G is feasible.

(2) **G is feasible ⇒ ReconstituteProgram succeeds.** ReconstituteProgram can fail in two places. First, it can fail in OrderRegion (see lines [3] and [5] of Figure 1) because all possible orderings lead to cyclic graphs. Second, if OrderRegion succeeds for all regions of G, ReconstituteProgram can still fail at the test in line [9] of Figure 1. Failure in either case means that G is infeasible. We shall show that if G is feasible then both tests will succeed; the proof is structured as follows:

(A) **G feasible ⇒ OrderRegion succeeds for all regions of G.** If G is feasible then for each region R of G, OrderRegion will find a total ordering for region R that respects at least one good total order (gto) of PDG G. See Section 4.1 for the proof of (2)(A).

(B) **G feasible ⇒ PDG identity test at line [9] of Figure 1 succeeds.** From (A) above we know that if G is feasible then a total order will be found for each region R of G such that each total order is the same as the total order on that region in some gto for G. Given this, we show that the total order found for G is indeed a gto.

By the Region Independence Lemma (Section 4.2), a total order t for a PDG is a good total order if the t-order for each region is the same as the order from *any* good total order of G for the corresponding region. This implies that the order chosen by OrderRegion over the whole PDG is a gto.

Since a gto is found for G by the first part of the algorithm and G_P is the PDG of the program produced from this gto, G must be identical to G_P. Therefore, if G is feasible then the test at line [9] of Figure 1 must succeed.

4.1. Proof that OrderRegion succeeds if G is feasible

DEFINITION (good total order): Let G be a feasible PDG. Let $t_1, t_2, ..., t_n$ be total left-to-right orders for the regions $R_1, R_2, ..., R_n$ of G. The set $t = \{t_1, t_2, ..., t_n\}$, is a *good total order* (gto) for G iff applying t to G's control-dependence subgraph yields an abstract-syntax tree that corresponds to program P, and G is P's program dependence graph.

DEFINITION (edge-set): The *edge-set* of a region R contains those edges, excluding loop-carried edges, whose source and target vertices are both in region R. During the course of ReconstituteProgram's execution, edges will be added to the edge-set of each region.

DEFINITION (graph G_C): ReconstituteProgram operates on the graph G_C, initially a copy of the PDG G, leaving the original PDG G intact. G_C is referred to as a graph instead of a PDG because ReconstituteProgram will modify it in trying to find a gto for PDG G. There is a close correspondence between the PDG G and the graph G_C, as the control-dependence subgraphs of the two are always identical during ReconstituteProgram's execution. Each region R in PDG G has a corresponding region in graph G_C. At times in the proof, we refer to the PDG G when we need to argue about a property of the original PDG. When we argue how ReconstituteProgram modifies the edge-sets of regions, we implicitly refer to the graph G_C.

DEFINITION (unconditional definition): In region R of feasible PDG G, vertex **d** is *an unconditional definition of variable x* if, for all gtos of G, all paths through the control-flow graph that corresponds to the control-dependence subtree rooted at **d** contain a definition of x.

DEFINITION (unconditional definition with respect to a use): Vertex **d** is *unconditional with respect to use* **u** if, in feasible PDG G, **d** is an *unconditional definition of x* such that at least one definition represented by **d** reaches use **u** and the control parent of **d** is a control ancestor of **u**.

The goal of OrderRegion is to find a partial ordering of the vertices of a region R that respects at least one good total order of G, such that any total order derived from this partial order by a topological sort respects a gto of G. To prove that OrderRegion succeeds if G is feasible we show that for every call to OrderRegion of the form "OrderRegion(G_C, R)", the six properties listed below hold. The first two properties concern the edge-set of region R when OrderRegion is first called; the remaining four properties concern edges added to R during this execution of OrderRegion.

(1) The edges in edge-set(R) that are also in G (*i.e.*, excluding edges in edge-set(R) added by previous calls to OrderRegion) respect all good total orders of G. This follows immediately from the fact that G is feasible.

(2) The edges in edge-set(R) that were added by previous calls to OrderRegion (*i.e.*, were added by Step (4) of procedure PreserveExposedUsesAndDefs – see Section 3.2) respect all good total orders of G. This follows from the Project-Edge Lemma of Section 4.1.1.2.

(3) The edges added to R by procedure PreserveExposedUsesAndDefs (called at line 1 of OrderRegion – see Figure 2) respect all good total orders of G. This follows from the Preserve-Exposed-Uses-And-Defs Lemma of Section 4.1.1.3.

(4) The edges added to R by PreserveSpans (called at line 2 of OrderRegion – see Figure 2) up to but not including its **do-od** loop (see Figure 7) respect all good total orders of G. This follows from the following observations: (a) PreserveSpans is called by OrderRegion immediately after the call to PreserveExposedUsesAndDefs; by point (3) above, the edge-set of R respects all good total orders

of G when PreserveSpans is first called. (b) The first step of PreserveSpans is to add edges so that the edge-set of R is transitively closed; this certainly cannot exclude any good total orders. (c) The final step of PreserveSpans before its **do-od** loop is to call OrderDependentSpans; the Order-Dependent-Spans Lemma of Section 4.1.2 ensures that the edges added in this step respect all good total orders.

(5) The edges added by the **do-od** loop of PreserveSpans respect at least one good total order of G. This follows from the Preserve-Spans Lemma of Section 4.1.3.

(6) The total ordering produced by the topological sort performed by OrderRegion (see Figure 2) respects at least one good total order of G. This follows from the Preserves-Spans-Orders-Representatives Lemma of Section 4.1.3.2.

4.1.1. Project-Edge and Preserve-Exposed-Uses-And-Defs Lemmas

4.1.1.1. Discussion of flow and def-order edges with respect to PreserveExposedUsesAndDefs

Consider a flow edge or def-order edge $v \rightarrow w$ in PDG G, where v and w are possibly in different regions. Let r be the least common control ancestor of v and w. Let $v_0..v_j$ ($j \geq 0$) be the control ancestors of v below r, where $v_j = v$. Let $w_0..w_k$ ($k \geq 0$) be the control ancestors of w below r, where $w_k = w$. Let v be a definition of variable x and let w be a definition or use of variable x, depending on whether $v \rightarrow w$ is a flow or def-order edge. If $v \rightarrow_f w$ is loop-carried then let s be the predicate vertex for the loop that carries the dependence (s may be r itself). Note that there is a separate edge for each loop that carries a dependence. Figure 8 shows the control dependence relationships described above.

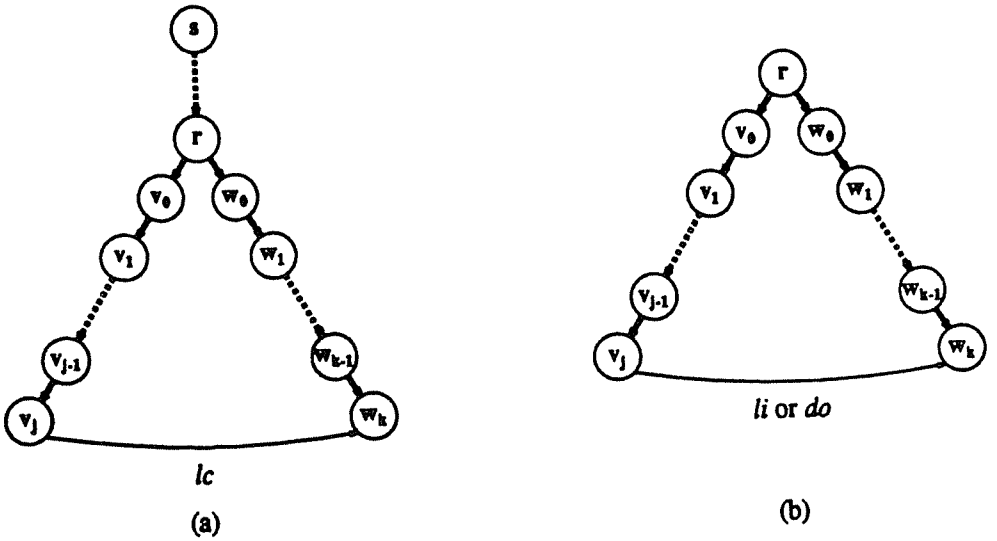

Figure 8. Edge $v \rightarrow w$ and related vertices. In (8)(a), edge $v \rightarrow w$ is loop carried; in (8)(b), edge $v \rightarrow w$ is either loop independent or def-order.

All the control ancestors of v_j are *representatives* of the definition of variable x at vertex v_j. Likewise, all the control ancestors of w_k are *representatives* of the use (or definition) of variable x at vertex w_k. A vertex may represent both many definitions and many uses.

A flow edge $v \rightarrow_f w$ implies that certain of v and w's control ancestors are upwards and/or downwards exposed in their enclosing regions. If $v \rightarrow_f w$ is:

- loop independent, then $v_1..v_j$ are all downwards exposed with respect to x in their enclosing regions, and $w_1..w_k$ are all upwards exposed with respect to x in their enclosing region.

- loop carried, then $v_\theta..v_j$ are all downwards exposed with respect to x in their enclosing regions, $w_\theta..w_k$ are all upwards exposed with respect to x in their enclosing regions, and furthermore, all vertices on the path from s to r, excluding s, are both upwards exposed and downwards exposed with respect to x in their enclosing regions.

The PreserveExposedUsesAndDefs algorithm breaks into four major steps:

(1) Identify upwards exposed uses.
(2) Identify downwards exposed definitions.
(3) Preserve exposed uses and definitions.
(4) Project edges onto the region head.

The upwards-exposed/downwards-exposed information (as defined by the previous paragraph) is propagated through the graph G_C by the bottom-up projection of flow edges and steps (1) and (2). We will not prove that this information is correctly propagated, as this is obvious from inspection of steps (1), (2) and (4) and the fact that PreserveExposedUsesAndDefs is applied bottom-up over the control-dependence subgraph of G_C.

Projection is the last step of the PreserveExposedUsesAndDefs algorithm. The projection step (applied bottom-up) accomplishes several tasks: Any flow or def-order edge ($v \rightarrow_f w$ or $v \rightarrow_{do} w$) will be projected up until its source and target representatives (v_θ and w_θ) have the same control parent, r. At this point, if the edge is loop-independent or def-order, then it remains in the edge-set of the region headed by r. However, if the edge is loop-carried by r then it is deleted from G_C. If the edge is loop-carried by a control ancestor of r, then the edge is projected onto r.

The edges that projection adds to the edge-set of region R are added before PreserveExposedUsesAndDefs(R) is called because PreserveExposedUsesAndDefs is applied bottom-up over the control-dependence subgraph of G_C and because the projection step always moves an edge's source (target) vertex from a vertex v to the control parent of v. The Project-Edge lemma defines the contents of the edge-set of R just before PreserveExposedUsesAndDefs(R) is invoked.

4.1.1.2. Project-Edge Lemma

LEMMA (Project-Edge). If G is feasible, then for all regions R, headed by vertex r, the edge-set of R (in graph G_C) will contain the edge $v_\theta \rightarrow w_\theta$ just before PreserveExposedUsesAndDefs(R) is invoked and forever after *iff*:

- r is the least common control ancestor of vertices v and w.
- There is a loop-independent or def-order edge v→w in PDG G.
- v_θ is a control ancestor of v and w_θ is a control ancestor of w.

Note that the edge $v_\theta \rightarrow w_\theta$ respects all gtos because all gtos must preserve the loop-independent or def-order edge $v \rightarrow w$. If w_θ were ordered before v_θ in region R then $v \rightarrow w$ could not be preserved in any total ordering of G, since w would be ordered before v.

PROOF: By step (4) of PreserveExposedUsesAndDefs, if the source (target) of a loop-independent or def-order edge is in region R and the target (source) is not in region R, then the region head of R becomes the source (target) of this edge. Since PreserveExposedUsesAndDefs is applied bottom-up over the control graph of G_C, any loop-independent or def-order edge $v \rightarrow w$ that satisfies the three criteria listed above will be projected up to become $v_\theta \rightarrow w_\theta$.

4.1.1.3. Preserve-Exposed-Uses-And-Defs Lemma

LEMMA (Preserve-Exposed-Uses-And-Defs). Given region R (headed by vertex r) of feasible PDG G such that the edge-set of R respects all gtos of G, the edges added by PreserveExposedUsesAndDefs(R) must respect all gtos of G.

PROOF: By contradiction. Let t be a gto of feasible PDG G. Let v and w be children of region head r such that v follows w in gto t. Suppose that PreserveExposedUsesAndDefs adds the edge $v \rightarrow_{peud} w$.[4] We shall consider the two reasons for which PreserveExposedUsesAndDefs may add this edge and show that each case leads to a contradiction:

(1) v represents an upwards exposed use of variable x, w represents a definition of x, and w is not a loop-independent flow predecessor of v.

Since v represents an upwards exposed use there must be an x-def-free execution path from the beginning of the region to v in gto t.[5] Since w precedes v in gto t, it is on this execution path. Therefore, there is an x-def-free execution path from w to v in gto t. Since the subtree rooted at w contains a definition of x, there must be a definition in that subtree that reaches v and so reaches the use of x in the subtree rooted at v in PDG G. Therefore, by projection, w must be a loop-independent flow predecessor of v under gto t, which contradicts the third point of (1).

(2) w represents a downwards exposed definition of variable x (d_1) and v represents a definition of x (d_2) such that there is no def-order edge from w to v.

Since v (d_2) follows w (d_1) in gto t and d_1 reaches a use outside of R, d_2 must also reach that same use. This means that the def-order edge $d_1 \rightarrow_{do} d_2$ must be in G. This edge will be projected up to become $w \rightarrow_{do} v$, the presence of which contradicts the last assumption of (2).

4.1.2. Order-Dependent-Spans Lemma

LEMMA (Order-Dependent-Spans). If G is feasible and the edges in region R's edge-set respect gto t, then any edges added to the edge-set by OrderDependentSpans must also respect gto t.

---

[4]Edges added to G_C by a procedure are noted using the procedure's "initials"; thus, $v \rightarrow_{peud} w$ denotes an edge added by procedure PreserveExposedUsesAndDefs, $v \rightarrow_{ps} w$ denotes an edge added by procedure PreserveSpans, *etc.*

[5]When we refer to "an execution path in a gto" we mean a path in the control-flow graph of the program defined by the gto.

PROOF: By contradiction. Suppose that OrderDependentSpans adds edge $c \rightarrow b$ to region R, but that in good total order t, b precedes c. OrderDependentSpans will add an edge from vertex c to vertex b in region R if there exist vertices a, v, w and variable x such that all the following conditions hold:

(1) $v \in \text{Span}(a,x)$, $w \in \text{Span}(b,x)$ and $v \rightarrow w \in \text{edge-set}(R)$.
(2) $c \in (\text{Span}(a,x) - \text{Span}(b,x))$.
(3) Edge $b \rightarrow_{do} a \notin \text{edge-set}(R)$.

Note that a and b cannot be the same vertex since $c \in (\text{Span}(a,x) - \text{Span}(b,x))$. Thus, there are two cases to consider: either b precedes a in gto t, or vice versa.

(1) b precedes a in gto t. Since v is in Span(a,x) and $v \rightarrow w$ respects gto t, w must follow a in gto t. Since (i) $b \rightarrow_{fl} w$, (ii) b precedes a in gto t, and (iii) w follows a in gto t, the definition of x at a must reach the use of x at w in gto t. This implies that w is in Span(a,x) as well as Span(b,x). Since the definitions of x at both a and b reach the use of x at w, and b precedes a in gto t, there must be a def-order edge $b \rightarrow_{do} a$ in G. However, according to condition (3) of OrderDependentSpans, $c \rightarrow b$ will not be added if the edge $b \rightarrow_{do} a$ exists.

(2) b follows a in gto t. Since b precedes c in gto t and the definition of x at a reaches the use of x at c, it must be that the definition of x at b reaches the use of x at c in gto t. This implies that $c \in \text{Span}(b,x)$, which means that $c \notin \text{Span}(a,x) - \text{Span}(b,x)$. Therefore, $c \rightarrow b$ would not be added to the edge-set of R by OrderDependentSpans since condition (2) is not met.

4.1.3. Preserve-Spans Lemma and related results

LEMMA (Preserve-Spans). If G is feasible then all independent x-spans of region R will be totally ordered by the addition of edges to R's edge-set in the loop of PreserveSpans. Furthermore, after the loop terminates, every topological ordering of R's edge-set respects a gto of PDG G.

PROOF: Since G is feasible there must be a way to order all independent x-spans so that the edge-set of R respects at least one gto of PDG G. As shown in [Horwitz87], the problem of ordering independent x-spans is NP-complete. The loop of PreserveSpans implements a backtracking algorithm to find an ordering of R's independent x-spans that respects at least one gto of G.

The loop terminates with success if all independent x-spans have been ordered and if the edge-set of R is acyclic. It is easy to show (by an inductive argument) that the loop will terminate with success if only correct choices are made when ordering R's independent x-spans. The base case for the induction is that 0 choices need to be made to totally order the independent x-spans. As shown previously, before the loop begins, the edge-set of R respects all gtos. Since no choices need to be made (there are no independent x-spans) and the edge-set of R is acyclic (since it respects all gtos), the loop must terminate with success.

The induction hypothesis is that after n choices have been made the edge-set of R respects at least one gto. The induction step (for $n+1$ choices) follows: Because there exist independent x-span pairs with heads h_1 and h_2, the algorithm must choose to add edge $h_1 \rightarrow h_2$ or $h_2 \rightarrow h_1$. Suppose the algorithm makes the correct choice - the added edge respects a gto. By the induction hypothesis, all edges in the edge-set of R before the choice was made respect at least one gto. Therefore, any paths created by the addition of the latest edge must also respect at least one gto. This implies that all edges added by AddEdgeAndClose will respect at least one gto. Therefore, by the OrderDependentSpans Lemma, the call to OrderDependentSpans can only add edges that respect this gto also.

Since there are a finite number of independent x-span pairs in region R, the loop will terminate with success if only correct choices are made.

The hard part of the proof is to show that if a bad choice is made (*i.e.*, one that creates a partial order that does not respect any gto), then a cycle will always arise at some later point in the execution of the loop (possibly many iterations later). The occurrence of a cycle in R's edge-set initiates the backtracking part of the PreserveSpans algorithm. If a cycle will always arise no matter what choices are made after a bad choice, the backtracking will, at some point, undo the bad choice and try the other choice, which *must* be correct. The following three sections address this part of the proof and are summarized below.

Section 4.1.3.1 addresses the following issue: Suppose t_1 is a gto for PDG G and t_2 is a bad total order (bto) such that t_2=t_1 except at a region headed by r, where $t_2(r)\neq t_1(r)$.[6] Let G' be the PDG of the program corresponding to t_2. For each flow edge $d \rightarrow_f u$ or def-order edge $d \rightarrow_{do} d'$ (with witness u) that is in one PDG but not the other, we show that the location of the vertices d, u and d′ must be restricted to certain portions of the PDG G. We then show that each flow or def-order edge that is in one PDG but not the other implies that a certain order exists in $t_2(r)$.

Section 4.1.3.1 characterizes the properties of a bad total order on a region. Section 4.1.3.2 uses the results of these sections to show that a cycle will arise in region R's edge-set if a bad choice is made by PreserveSpans. Once a bad choice has been made, the partial order of R's edge-set, $t_p(r)$, cannot respect any gto. Let $t_2(r)$ be a total order that respects the partial order $t_p(r)$. $t_2(r)$ cannot respect any gto. This implies that $t_2(r)$ (and thus $t_p(r)$) has one of the ordering mistakes enumerated in Section 4.1.3.1. In each of the ordering cases that can arise, we show that if PreserveSpans orders the remaining independent x-spans of the region, a cycle will arise in the edge-set of R no matter what choices are made.

4.1.3.1. Properties of a Bad Total Order

Let t_1 be a gto of feasible PDG G. Construct total order t_2 as follows: t_2 is the same as t_1 except at the ordering of vertex r's children, where $t_2(r)$ does not correspond to any gto. Let G' be the PDG of the program corresponding to t_2. Since t_2 is a bto there must either be: (1) An edge $d \rightarrow_f u$ that is in one PDG but not the other, or (2) an edge $d \rightarrow_{do} d'$ that is in one PDG but not the other. Assume that d and d′ are definitions of the variable x and that u is a use of the variable x. We first show that in each case, the locations of d and u (or d and d′) are restricted to certain parts of the PDG G. Next, for each possible location for d and u (or d and d′) we characterize the possible ordering mistakes that occur in $t_2(r)$.

Location of d, u (or d, d′)

(1) Let d and u be the source and target of edge $d \rightarrow_f u$, which is one PDG but not the other. One of the following must be true of d and u:

(A) d is a control descendant of r; u is not a control descendant of r.
(B) u is a control descendant of r; d is not a control descendant of r.
(C) d and u are both control descendants of r and there exist control children c_1 and c_2 of r such that $c_1 \neq c_2$, c_1 is a control ancestor of d, and c_2 is a control ancestor of u.

[6] t(r) is the left-to-right total order of the control children of region head r in total order t. Note that the edge-set of a region headed by r is similar to t(r) in that it determines an ordering (possibly total) on the control children of r.

Proof of (1). We show that part (1) must hold by assuming that $\mathbf{d} \rightarrow_f \mathbf{u}$ is in one PDG but not the other and then showing that such an edge could not exist in the locations *not* listed in part (1). It is trivial to show, by example, cases where (A), (B) and (C) arise and we leave them to the reader to generate. If neither (A) nor (B) nor (C) hold, then either (i) neither **d** nor **u** is a control descendant of **r**, or (ii) **d** and **u** are both control descendants of **r** and there exists a control child **c** of **r** such that both **d** and **u** are control descendants of **c**.

(i) Neither **d** nor **u** is a control descendant of **r**.

Without loss of generality, assume the edge $\mathbf{d} \rightarrow_f \mathbf{u}$ is in G, but not in G'. We show that $\mathbf{d} \rightarrow_f \mathbf{u}$ must be in G'.

If **r** is not on an execution path that gives rise to $\mathbf{d} \rightarrow_f \mathbf{u}$ then $\mathbf{d} \rightarrow_f \mathbf{u}$ must be in G' since $t_1 = t_2$ outside the subtree rooted at **r**. If **r** is on an execution path that gives rise to $\mathbf{d} \rightarrow_f \mathbf{u}$ then we must consider two cases:

- **r** is a **while**-predicate. **r** may evaluate to **false**, in which case, **r**'s children will not execute. This implies that there is an x-def-free execution path from **d** to **u** in both t_1 and t_2 that does not enter the region headed by **r**. Thus, $\mathbf{d} \rightarrow_f \mathbf{u}$ must be in G'.

- **r** is an **if**-predicate. Either the **false** region or the **true** region of **r** must contain no unconditional definitions of x. Without loss of generality, suppose the **true** region contains no unconditional definitions of x. No matter what the order of $t_2(r)$, there must be an x-def-free execution path through the subtree rooted at **r**. Since $t_1 = t_2$ outside the subtree rooted at **r**, $\mathbf{d} \rightarrow_f \mathbf{u}$ must be in G'.

(ii) **d** and **u** are both control descendants of **r** and there exists a control child **c** of **r** such that both **d** and **u** are control descendants of **c**. There are two cases to consider:

- $\mathbf{d} \rightarrow_f \mathbf{u}$ is loop-independent or is loop-carried by **c** or a control descendant of **c**

  Since $t_1 = t_2$ inside the subtrees rooted at children of **r**, any x-def-free execution path from **d** to **u** in t_1 must be present in t_2 and *vice versa*. Therefore, $\mathbf{d} \rightarrow_f \mathbf{u}$ must be in both G and G' in this case.

- $\mathbf{d} \rightarrow_f \mathbf{u}$ is loop-carried by **r** or a control ancestor of **r**.

  Since $t_1 = t_2$ inside the subtree rooted at **c**, **d** is potentially downwards exposed[7] in subtree **c** and **u** is likewise upwards exposed in subtree **c** in both t_1 and t_2. Also, $t_1 = t_2$ outside of the subtree rooted at **r**, so any x-def-free execution path from the point after **r** to the point before **r** in t_1 will be present in t_2 (and *vice versa*). Since $\mathbf{d} \rightarrow_{lc} \mathbf{u}$ is in G (or G') the region headed by **r** must not contain any unconditional definitions. Therefore, $\mathbf{d} \rightarrow_{lc} \mathbf{u}$ must exist in both G and G' if it exists in either. This contradicts our assumption that $\mathbf{d} \rightarrow_f \mathbf{u}$ exists in one PDG but not the other.

(2) Let **d** and **d'** be the source and target of edge $\mathbf{d} \rightarrow_{do(u)} \mathbf{d'}$, which is one PDG but not the other. We

[7] A definition is "potentially downwards exposed" in code segment C if it reaches the end of C. Note that an ordered control-dependence subtree corresponds to a code segment, so that the term "potentially downwards exposed" can be used to refer to a definition in a control-dependence subtree under a particular order.

shall show that either **d** and **d′** are both control descendants of **r** or that one of **d**, **d′** or common use **u** is a control descendant of **r**.

(A) Assume that $\mathbf{d} \longrightarrow_f \mathbf{u}$ and $\mathbf{d}' \longrightarrow_f \mathbf{u}$ are in both PDGs. Since both G and G' are feasible, both have def-order edges between **d** and **d′**. By assumption, the edge $\mathbf{d} \longrightarrow_{do} \mathbf{d}'$ is one PDG but not the other. Without loss of generality, suppose $\mathbf{d} \longrightarrow_{do} \mathbf{d}'$ is in G, but not in G'. There must be a def-order edge $\mathbf{d}' \longrightarrow_{do} \mathbf{d}$ in G' if this is so. This means **d′** precedes **d** in t_2 and **d′** follows **d** in t_1. Since t_2 is equal everywhere to t_1 except at $t_2(\mathbf{r})$, it must be that **d** and **d′** are control descendants of **r** and that $t_2(\mathbf{r})$ reversed their order from $t_1(\mathbf{r})$.

(B) Assume that $\mathbf{d} \longrightarrow_f \mathbf{u}$ and $\mathbf{d}' \longrightarrow_f \mathbf{u}$ are not in both PDGs. By part (1) of the proof, one of **d**, **d′** or **u** must be a control descendant of **r**.

Ordering mistakes in t_2

We have established that if t_2 is a bad total ordering then (1) there is an edge e that is in G but not in G' or *vice versa*, and (2) there are certain restrictions on the locations of the endpoints of e. We now establish facts about t_2, using a case analysis on edge e with subcases on the location of e's endpoints; these facts will be used to prove the correctness of OrderRegion's backtracking algorithm.

(1) Edge e is a flow edge $\mathbf{d} \longrightarrow_f \mathbf{u}$.

(1)(A) Suppose **d** is a control descendant of **r**, **c** is the child of **r** that is a control ancestor of **d**, and **u** is not a control descendant of **r**.

Since $t_1 = t_2$ inside the subtree rooted at **c**, **d** must be potentially downwards exposed in that subtree in both t_1 and t_2. Since $t_1 = t_2$ outside the subtree rooted at **r**, there must be an x-def-free execution path from the point after **r** to **u** in both t_1 and t_2. Therefore, if:

(i) $\mathbf{d} \longrightarrow_f \mathbf{u}$ in G, not in G', then $t_2(\mathbf{r})$ must order an unconditional definition of x after **c**.
(ii) $\mathbf{d} \longrightarrow_f \mathbf{u}$ in G', not in G, then $t_2(\mathbf{r})$ must order all unconditional definitions of x before **c**.

(1)(B) Suppose **u** is a control descendant of **r**, **c** is the child of **r** that is a control ancestor of **u**, and **d** is not a control descendant of **r**.

Since $t_1 = t_2$ inside the subtree rooted at **c**, **u** must be upwards exposed in that subtree in both t_1 and t_2. Since $t_1 = t_2$ outside the subtree rooted at **r**, there must be an x-def-free execution path from the point after **d** to before **r** in both t_1 and t_2. Therefore, if:

(i) $\mathbf{d} \longrightarrow_f \mathbf{u}$ in G, not in G', then $t_2(\mathbf{r})$ must order an unconditional definition of x before **c**.
(ii) $\mathbf{d} \longrightarrow_f \mathbf{u}$ in G', not in G, then $t_2(\mathbf{r})$ must order all unconditional definitions of x after **c**.

(1)(C) Suppose **d** and **u** are both control descendants of **r**, $\mathbf{c}_1$ is the control child of **r** that is a control ancestor of **d**, $\mathbf{c}_2$ is the control child of **r** that is a control ancestor of **u**, and $\mathbf{c}_1 \neq \mathbf{c}_2$.

Since $t_1 = t_2$ inside the subtrees rooted at $\mathbf{c}_1$ and $\mathbf{c}_2$, **d** must be potentially downwards exposed in subtree $\mathbf{c}_1$ and **u** must be upwards exposed in subtree $\mathbf{c}_2$ in both t_1 and t_2.

NOTATION. The notation $\mathbf{v} \longrightarrow^+ \mathbf{w}$ denotes that $t_2(\mathbf{r})$ orders vertex **v** before vertex **w**, where both **v** and **w** are control children of **r**.

(i) $d \longrightarrow_f u$ in G, not in G'. Consider the type of $d \longrightarrow_f u$:

(a) $d \longrightarrow_f u$ is loop independent. Either there exists an unconditional definition d' of x and $t_2(r)$ ordered $c_1 \longrightarrow^+ d' \longrightarrow^+ c_2$, or $t_2(r)$ ordered $c_2 \longrightarrow^+ c_1$.

(b) $d \longrightarrow_f u$ is loop carried. An unconditional definition d' must exist in this case. (Why? Since $t_1 = t_2$ outside of the subtree rooted at r, there must be an x-def-free path from the point after r to the point before r that goes around the loop that carries $d \longrightarrow_{lc} u$ in both t_1 and t_2. If there are no unconditional definitions of x under r, $d \longrightarrow_{lc} u$ must be in both PDGs.) $t_2(r)$ must order d' after c_1 ($c_1 \longrightarrow^+ d'$) or must order d' before c_2 ($d' \longrightarrow^+ c_2$).

(ii) $d \longrightarrow_f u$ in G', not in G. Here, we consider whether $t_2(r)$ orders c_1 before c_2 or after it.

(a) $t_2(r)$ orders $c_1 \longrightarrow^+ c_2$. $t_2(r)$ must also order all unconditional definitions of x (if any) either before c_1 or after c_2. Note that in this case the edge is loop-independent. If it were loop-carried and there were an unconditional definition of x under r then $d \longrightarrow_f u$ could not be in G', as assumed.

(b) $t_2(r)$ orders $c_2 \longrightarrow^+ c_1$. $t_2(r)$ must also order all unconditional definitions of x between c_2 and c_1. Note that in this case the edge is loop-carried.

(2) Edge e is a def-order edge $d \longrightarrow_{do} d'$.

(2)(A) Suppose $d \longrightarrow_f u$ and $d' \longrightarrow_f u$ are in both PDGs. Then, as shown above, d and d' must be control descendants of r such that $t_2(r)$ reverses the order of d and d' from $t_1(r)$.

(2)(B) Suppose $d \longrightarrow_f u$ and $d' \longrightarrow_f u$ are not in both PDGs. Then $t_2(r)$ must have ordered according to (1)(A), (1)(B) or (1)(C).

4.1.3.2. A bad choice in PreserveSpans leads to a cycle

We now show that if PreserveSpans makes a bad choice, then a cycle will arise in the edge-set of R no matter what choices are made (by PreserveSpans) after the bad choice. The argument is by contradiction: assume that PreserveSpans makes a bad choice and finishes without a cycle arising.

Since PreserveSpans made a bad choice, the partial order $t_p(r)$ induced by PreserveSpans cannot respect any gto of PDG G. Furthermore, *no* total ordering of r's children that respects $t_p(r)$ can be part of a gto.

A bad total order for R must have one of the ordering mistakes enumerated in Section 4.1.3.1. Each one of these ordering mistakes requires the ordering of an unconditional definition with respect to other representative definitions and uses of the same variable, or requires that the order of a representative definition and use be switched. By the following lemma, after PreserveSpans terminates, all representative definitions are ordered with respect to all other representative definitions and uses of the same variable. This implies that any ordering mistake (as outlined in Section 4.1.3.1) present in any total order respecting $t_p(r)$ must be present in $t_p(r)$.

LEMMA (Preserve-Spans-Orders-Representatives). If, for region R, PreserveSpans terminates without a cycle arising, then for every pair of vertices v, w, such that v and w both represent definitions of variable x or v represents a definition of x and w represents a use of x, R's edge-set includes either a path $v \longrightarrow^+ w$ or $w \longrightarrow^+ v$.

PROOF: Any pair of vertices, v and w, that both represent the definition of variable x must be ordered with respect to each other since both vertices head x-spans and PreserveSpans totally orders all x-spans.

Any pair of vertices, v and w, such that v represents a definition of x and w represents a use of x must be ordered with respect to one another by the following argument:

Either w is a member of v's span or not. If so, then there is an edge $v \rightarrow_{li} w$. If not, then w is either upwards exposed in R or is reached by unconditional definition d', where d' is a child of r. If w is upwards exposed in R then (since there is no edge $d' \rightarrow_{li} w$) there will be an edge $w \rightarrow_{pseud} v$ added by PreserveExposedUsesAndDefs. If w is reached by unconditional definition d', then there is an edge $d' \rightarrow_{li} w$. Since v and d' both head x-spans, PreserveSpans will either add edge $v \rightarrow_{ps} d'$ or edge $w \rightarrow_{ps} v$. In the latter case, w is directly ordered with respect to v. In the former case, w is ordered with respect to v by $v \rightarrow_{ps} d' \rightarrow_{li} w$. □

Let t_1 be a gto of G. Construct total order t_2, where $t_2 = t_1$ except at $t_2(r)$ where $t_2(r)$ = a total ordering respecting $t_p(r)$. Let G' be the PDG corresponding to total order t_2. Since t_2 is a bad total order, $G \neq G'$. We show that a cycle must have arisen in PreserveSpans by a case analysis on the differences in the flow-edge sets of G and G'. Each case implies that $t_2(r)$ contains a particular ordering mistake, which in turn implies (as argued previously) that $t_p(r)$ has the same ordering mistake. We show that the presence of this ordering in $t_p(r)$ will always lead to a cycle in PreserveSpans. For each case, the ordering mistake that arises corresponds to the one enumerated in the corresponding part of Section 4.1.3.1.

(1) Edge e is a flow edge $d \rightarrow_f u$.

(1)(A) Suppose d is a control descendant of r, c is the control child of r that is a control ancestor of d, and u is not a control descendant of r.

(i) $d \rightarrow_f u$ in G, not in G'. $t_p(r)$ must order an unconditional definition d' after c by path $c \rightarrow^+ d'$. Since c is identified as downwards exposed in the region headed by r, PreserveExposedUsesAndDefs would have added $d' \rightarrow_{pseud} c$ unless $c \rightarrow_{do} d'$. In feasible PDGs, c cannot both be downwards exposed in the region headed by r and the source of a def-order edge to an unconditional definition in the region. Therefore, $c \rightarrow_{do} d'$ does not exist and a cycle exists by $d' \rightarrow_{pseud} c \rightarrow^+ d'$.

(ii) $d \rightarrow_f u$ in G', not in G. For each unconditional definition d' (there must be at least one), $t_p(r)$ must order d' before c by path $d' \rightarrow^+ c$. $t_1(r)$ must order at least one unconditional definition d'' after c (otherwise (ii) could not arise). Therefore, there must be some unconditional definition d'', a child of r, such that $d'' \rightarrow_f u$ is in G. Since d'' is downwards exposed by this edge, PreserveExposedUsesAndDefs would have added edge $c \rightarrow_{pseud} d''$, unless $d'' \rightarrow_{do} c$. Since c precedes d'' in $t_1(r)$, $d'' \rightarrow_{do} c$ cannot exist in G. A cycle exists by $c \rightarrow_{pseud} d'' \rightarrow^+ c$.

(1)(B) Suppose u is a control descendant of r, c is the control child of r that is a control ancestor of u, and d is not a control descendant of r.

(i) $d \rightarrow_f u$ in G, not in G'. $t_p(r)$ must order an unconditional definition d' before c by path $d' \rightarrow^+ c$. Since c is identified as upwards exposed in the region headed by r, PreserveExposedUsesAndDefs would have added $c \rightarrow_{pseud} d'$ unless $d' \rightarrow_{li} c$. $d' \rightarrow_{li} c$ could not be in PDG G since $d \rightarrow_f u$ is in G. Therefore, a cycle exists by $d' \rightarrow_{pseud} c \rightarrow^+ d'$.

(ii) $d \rightarrow_f u$ in G', not in G. For each unconditional definition d' (there must be at least one), $t_p(r)$ must order d' after c by path $c \rightarrow^+ d'$. $t_1(r)$ must order at least one unconditional definition d'' before c (otherwise (ii) could not arise). This implies that $d'' \rightarrow_{li} u$ is in G. Since $d'' \rightarrow_{li} u$ and $c \rightarrow^+ d'$ (for all d'), a cycle exists, as $d'' \rightarrow_{li} u$ will project up to be $d'' \rightarrow_{li} c$.

(1)(C) Suppose **d** and **u** are both control descendants of **r**, c_1 is the representative of **d** in the region headed by **r**, c_2 is the representative of **u** in the region headed by **r**, and $c_1 \neq c_2$.

(i) $d \rightarrow_f u$ in G, not in G'. There are four subcases to consider:

(a) Suppose $d \rightarrow_f u$ is loop independent, there exists unconditional definition d', and $t_p(r)$ orders by $c_1 \rightarrow^+ d' \rightarrow^+ c_2$. $c_1 \rightarrow_{li} c_2$ exists by projection. Since c_1 and d' head dependent x-spans, OrderDependentSpans must order them. When OrderDependentSpans considers $c_1 \rightarrow^+ d'$ it will order Span(c_1,x) before Span(d',x). c_2 is in Span(c_1,x), but not in Span(d',x) (if there were a loop-independent edge from d' to c_2 then there would be a def-order edge either from c_1 to d' or *vice versa*; $c_1 \rightarrow_{do} d'$ cannot exist since d' is unconditional; $d' \rightarrow_{do} c_1$ creates the cycle $d' \rightarrow_{do} c_1 \rightarrow^+ d'$). Since c_2 is not in Span(d',x), OrderDependentSpans will add $c_2 \rightarrow_{ps} d'$, creating the cycle $c_2 \rightarrow_{ps} d' \rightarrow^+ c_2$.

(a') Suppose $d \rightarrow_f u$ is loop independent and $t_p(r)$ orders by $c_2 \rightarrow^+ c_1$. A cycle exists since we have $c_1 \rightarrow_{li} c_2$ in $t_p(r)$ by projection.

(b) Suppose $d \rightarrow_f u$ is loop carried, there exists unconditional definition d', and $t_p(r)$ orders by $c_1 \rightarrow^+ d'$. c_1 is downwards exposed in the region headed by **r**. In this case, the argument is the same as in (1)(A)(i).

(b') Suppose $d \rightarrow_f u$ is loop carried, there exists unconditional definition d', and $t_p(r)$ orders by $d' \rightarrow^+ c_2$. c_2 is upwards exposed in the region headed by **r**. In this case, the argument is the same as in (1)(B)(i).

(ii) $d \rightarrow_f u$ in G', not in G. There are two subcases to consider:

(a) $t_p(r)$ orders by $c_1 \rightarrow^+ c_2$. $t_p(r)$ must also order each unconditional definition d' either before c_1 or after c_2. As noted in Section 4.1.3.1, $d \rightarrow_f u$ must be loop independent in this case. Because G is feasible, either c_2 is upwards exposed in the region headed by **r** or there exists unconditional definition d', where d' is a control child of **r**, such that $d' \rightarrow_{li} c_2$. Suppose the former case: PreserveExposedUsesAndDefs would have added edge $c_2 \rightarrow_{pseud} c_1$ since there could be no edge $c_1 \rightarrow_{li} c_2$ (if such an edge existed then $d \rightarrow_f u$ would have to be in G). This forms the cycle $c_2 \rightarrow_{pseud} c_1 \rightarrow^+ c_2$.

Suppose the latter case, there exists unconditional definition d' such that $d' \rightarrow_{li} c_2$ is in G. Suppose $d'=c_1$. This is impossible; the fact that $d \rightarrow_{li} u$ is in G and not in G implies that there can be *no* flow dependence from the subtree rooted at c_1 to the subtree rooted at c_2 in G. If there were, then $d \rightarrow_{li} u$ would be in G.

Suppose $d' \neq c_1$ Now, $t_p(r)$ must either order d' after c_2 or before c_1. If $t_p(r)$ orders by $c_2 \rightarrow^+ d'$ then there is a loop immediately. Instead, suppose $t_p(r)$ orders by $d' \rightarrow^+ c_1$. Since d' and c_1 both head dependent x-spans, OrderDependentSpans will order them. When OrderDependentSpans considers edge $d' \rightarrow^+ c_1$, it will order Span(d',x) before Span(c_1,x), which means it will add the edge $c_2 \rightarrow_{ps} c_1$ (since c_2 can not be in Span(c_1,x)). This forms the cycle $c_2 \rightarrow_{ps} c_1 \rightarrow^+ c_2$.

(b) $t_p(r)$ orders by $c_2 \rightarrow^+ c_1$. For all unconditional definitions d', $t_p(r)$ must order by $c_2 \rightarrow^+ d' \rightarrow^+ c_1$. Since $d \rightarrow_f u$ is not in G, $t_1(r)$ must order an unconditional definition d'' either after c_1 or before c_2. In the first case the argument for a cycle arising is the same as in (1)(A)(ii). In the second case the argument for a cycle arising is the same as in (1)(B)(ii).

(2) $d \rightarrow_a d'$ in G (G'), not in G' (G).

(2)(A) $d \rightarrow_f u$ and $d' \rightarrow_f u$ are in both PDGs. Without loss of generality, assume that $d \rightarrow_{do} d'$ is in G but not in G'. $t_p(r)$ must reverse the order of d and d' from $t_1(r)$ by $d' \rightarrow^+ d$. This edge goes in the opposite direction of $d \rightarrow_{do} d'$ and creates a cycle.

(2)(B) $d \rightarrow_f u$ and $d' \rightarrow_f u$ are not in both PDGs. $t_p(r)$ must have ordered according to (1)(A), (1)(B) or (1)(C), which, as already shown, creates a cycle.

4.2. Region-Independence Lemma

LEMMA (Region-Independence). Let G be a feasible PDG. Let t_1 and t_2 be distinct good total orders of G. Create total order t_3 as follows: t_3 is the same as t_1, except at a region R of G, headed by vertex r, where $t_3(r) = t_2(r)$. Then t_3 is a good total order.

PROOF: Let G' be the PDG of the program corresponding to t_3. We will show that G and G' are identical.

It is obvious that the vertex sets and control dependence edges of G and G' must be identical. We first show that the set of loop-independent and loop-carried flow edges in the two graphs must be identical. Showing that the sets of def-order edges are identical follows easily once this has been done.

4.2.1. Loop-independent and loop-carried flow edges identical

The following properties of good total orders simplify this part of the proof:

(P1) Let d be a control child of r that represents a definition of x. Let u be a use of x such that $d \rightarrow_f u$. Let t be a gto. If t orders d before u, then let D_t be the set of vertices that are control children of r and are ordered after d in t(r), but before u. If t orders u before d, then let D_t be the set of vertices that are control children of r and are either ordered after d or before u in t(r). None of the vertices in D_t can be an unconditional definition of x if t is a gto. Otherwise, $d \rightarrow_f u$ would not be preserved.

(P2) Let u be a control child of r that represents a use of x. Let d be a definition of x such that $d \rightarrow_f u$. Let t be a gto. If t orders d before u, then let U_t be the set of vertices that are control children of r and are ordered after d but before u in t(r). If t orders u before d, then let U_t be the set of vertices that are control children of r and are ordered either after d or before u in t(r). None of the vertices in U_t can be an unconditional definition of x if t is a gto. Otherwise, $d \rightarrow_f u$ would not be preserved.

(P3) If there is a flow edge $d \rightarrow_f u$ in G such that d and u are control descendants of r (with representatives c_1 and c_2 in the region headed by r) then the relative order of c_1 and c_2 must be the same in t(r), for all gtos t.

(P4) Let u represent a use of x. If u is upwards exposed in region R in good total order t_1, then u is upwards exposed in region R in *all* good total orders.

Let d (a definition of x) be the source of the flow edge under consideration and u (a use of x) be the target. We break the proof into two parts: (1) the presence of $d \rightarrow_f u$ in G implies that $d \rightarrow_f u$ is in G'; (2) the presence of $d \rightarrow_f u$ in G' but not in G leads to a contradiction. For each part, we do a case analysis on the location of d and u with respect to vertex r.

(1) $d \rightarrow_f u$ in $G \Rightarrow d \rightarrow_f u$ in G'.

(A) Suppose both **d** and **u** are outside the subtree rooted at **r**.

If **r** is not on an execution path that gives rise to $\mathbf{d} \rightarrow_f \mathbf{u}$ then $\mathbf{d} \rightarrow_f \mathbf{u}$ must be in G' since $t_1 = t_3$ outside the subtree rooted at **r**. If **r** is on an execution path that gives rise to $\mathbf{d} \rightarrow_f \mathbf{u}$ then we must consider two cases:

- **r** is a while-predicate. **r** may evaluate to **false**, in which case, **r**'s children will not execute. This implies that there is an x-def-free execution path from **d** to **u** in both t_1 and t_3 that does not enter the region headed by **r**. Thus $\mathbf{d} \rightarrow_f \mathbf{u}$ must be in G'.

- **r** is an if-predicate. Either the **false** region or the **true** region of **r** must contain no unconditional definitions of x. Without loss of generality, suppose the **true** region has no unconditional definitions of x. No matter what the order of $t_2(\mathbf{r})$, there must be an x-def-free execution path through the subtree rooted at **r**. Since $t_1 = t_3$ outside the subtree rooted at **r**, $\mathbf{d} \rightarrow_f \mathbf{u}$ must be in G'.

(B) Suppose **d** is inside the subtree rooted at **r** and **u** is outside. Let **c** be the control child of **r** that is a control ancestor of **d**.

d and **u** must be in same relative order in t_1 and t_3. **d** must be potentially downwards exposed in the subtree rooted at **c** under order t_3 since $t_3 = t_1$ for this subtree. By (P1), no vertex to the right of **c** in $t_2(\mathbf{r})$ may be an unconditional definition of x. Therefore, the definition of x at **d** reaches the point after **r** in t_3, as $t_3(\mathbf{r}) = t_2(\mathbf{r})$. There must be an x-def-free execution path from the point after **r** to **u** in t_3 since $t_3 = t_1$ for all regions outside the subtree rooted at **r**. Therefore, **d** must reach **u** in t_3 and $\mathbf{d} \rightarrow_f \mathbf{u}$ is in G'.

(C) Suppose **u** is inside the subtree rooted at **r** and **d** is outside. Let **c** be the control child of **r** that is a control ancestor of **u**.

d and **u** must be in same relative order in t_1 and t_3. **u** must be upwards exposed in the subtree rooted at **c** under order t_3 since $t_3 = t_1$ for this subtree. By (P2), no vertex to the left of **c** in $t_2(\mathbf{r})$ may be an unconditional definition of x since t_2 is a gto. Therefore, there is an x-def-free execution path from the point before **r** to **u** in t_3. There must be an x-def-free execution path from the point after **d** to the point before **r** since $t_3 = t_1$ for all regions outside the subtree rooted at **r**. Therefore, **d** must reach **u** in t_3 and $\mathbf{d} \rightarrow_f \mathbf{u}$ is in G'.

(D) Suppose **d** and **u** are both inside the subtree rooted at **r**. Let $\mathbf{c}_1$ be the control child of **r** that is a control ancestor of **d**, and let $\mathbf{c}_2$ be the control child of **r** that is a control ancestor of **u**. $\mathbf{c}_1$ and $\mathbf{c}_2$ may or may not be the same vertex.

If $\mathbf{c}_1$ and $\mathbf{c}_2$ are distinct vertices, then by (P3) they are in the same relative order under t_1 and t_2, and thus they are in the same relative order under t_1 and t_3. If $\mathbf{c}_1$ and $\mathbf{c}_2$ are the same vertex, then **d** and **u** are in the same relative order under t_1 and t_3 since $t_1 = t_3$ in this subtree.

An execution path in a gto giving rise to $\mathbf{d} \rightarrow_f \mathbf{u}$ may pass through three areas of the PDG: first, inside subtrees rooted at the children of **r**, second, in the region headed by **r**, and third, outside the subtree rooted at **r**. For each of these areas, we show that t_1 and t_3 have a corresponding x-def-free execution path from **d** to **u**.

Any part of an execution path (giving rise to $\mathbf{d} \rightarrow_f \mathbf{u}$) interior to a subtree rooted at a child of **r** will be x-def-free in both t_1 and t_3 since $t_1 = t_3$ for this part of the PDG. By (P1) and (P2), that part of an execution path (giving rise to $\mathbf{d} \rightarrow_f \mathbf{u}$) that contains children of **r** may not contain any unconditional definition to x in either $t_1(\mathbf{r})$ or $t_2(\mathbf{r})$. Any part of an execution path outside of the subtree rooted at **r** will be x-def-free in both t_1 and t_3 since $t_1 = t_3$ in this part of the PDG. Therefore, $\mathbf{d} \rightarrow_f \mathbf{u}$ must be in G'.

(2) $\mathbf{d} \rightarrow_f \mathbf{u}$ in G', not in $G \Rightarrow$ contradiction.

(A) Suppose **d** and **u** are outside the subtree rooted at **r**.

By an argument similar to that of (1)(A), we can show that $\mathbf{d} \rightarrow_f \mathbf{u}$ in G' implies that $\mathbf{d} \rightarrow_f \mathbf{u}$ is in G.

(B) Suppose **d** is inside the subtree rooted at **r** and **u** is outside. Let **c** be the control child of **r** that is a control ancestor of **d**.

There must exist an x-def-free path from the point after **r** to **u** in t_1, and from the point after **d** to the point after **c** in t_1 (since t_3=t_1 for all regions except the one headed by **r** and since **d** reaches **u** in G'). Since there is no edge $\mathbf{d} \rightarrow_f \mathbf{u}$ in G, it must be that $t_1(\mathbf{r})$ orders an unconditional definition **d′** after **c** but that the order $t_2(\mathbf{r})$ does not place any unconditional definitions after **c**. Under order $t_1(\mathbf{r})$, there must be a flow edge from **d′** to **u** in G (or from some other unconditional definition **d″** that t_1 orders after **d′**. Without loss of generality, assume the edge is from **d′**.)

Suppose **d** is a child of **r**. This implies that **d** kills **d′** in order $t_2(\mathbf{r})$, which means that $\mathbf{d}' \rightarrow_f \mathbf{u}$ is in G but not in G'. This contradicts our proof of part (1). Suppose **d** is not a child of **r** but resides in a subtree rooted at **c**, which is a child of **r**. There must exist a definition **d″** in subtree **c** that reaches **u** in t_2, since **d′** reaches **u** and all unconditional definitions precede **c** in $t_2(\mathbf{r})$. This implies that there is a def-order edge $\mathbf{d}' \rightarrow_{do} \mathbf{d}''$ in G. However, $t_1(\mathbf{r})$ does not respect this def-order edge as it orders **d′** after **c**. Therefore, t_1 must be a bad total order. Contradiction.

(C) Suppose **u** is inside subtree rooted at **r** and **d** is outside. Let **c** be the control child of **r** that is a control ancestor of **u**.

There must exist an x-def-free path from the point after **d** to before **r** in t_1, and from the point before **c** to the point before **u** in t_1 since $t_3 = t_1$ for all regions except R. Since there is no edge $\mathbf{d} \rightarrow_f \mathbf{u}$ in G, it must be that $t_1(\mathbf{r})$ placed an unconditional definition **d′** before **c** but that the order $t_2(\mathbf{r})$ does not place any unconditional definitions before **c**. Without loss of generality, assume there are no other unconditional definitions between **d′** and **c** in $t_1(\mathbf{r})$. Under order $t_1(\mathbf{r})$, there must be a loop-independent flow edge from **d′** to **u** in G. This implies that $t_2(\mathbf{r})$ is a bto since it places **d′** after **c**.

(D) Suppose both **d** and **u** are inside the subtree rooted at **r**.

(i) **d** and **u** are in the same subtree rooted at **c**, a control child of **r**.

If $\mathbf{d} \rightarrow_f \mathbf{u}$ is a loop-independent edge then it must be in both G and G' since the order for subtree **c** is the same under t_1 and t_3. If $\mathbf{d} \rightarrow_f \mathbf{u}$ is a loop-carried edge then we must consider two cases:

- $\mathbf{d} \rightarrow_{lc} \mathbf{u}$ is carried by vertex **r** or a control ancestor of **r**.

  d is potentially downwards exposed in subtree **c** and **u** is upwards exposed in subtree **c** in both t_1 and t_3 since $t_1 = t_3$ for these subtrees. Since $\mathbf{d} \rightarrow_{lc} \mathbf{u}$ is in G', there can be no unconditional definition of x in region R outside the subtree rooted at **c**. Thus, any difference between $t_1(\mathbf{r})$ and $t_2(\mathbf{r})$ is irrelevant, and $\mathbf{d} \rightarrow_{lc} \mathbf{u}$ must be in G.

- $\mathbf{d} \rightarrow_{lc} \mathbf{u}$ is carried by **c** or a control descendant of **c**.

  $\mathbf{d} \rightarrow_{lc} \mathbf{u}$ must be in G if it is in G' since the order for subtree **c** is the same in orders t_1 and t_3 and the edge is carried by **c** or a control descendant of **c**.

(ii) d and u are in different subtrees (c_1 and c_2)

There are two cases to consider:

• $d \rightarrow_f u$ is loop independent. Since $d \rightarrow_{li} u$ is not in G, $t_1(r)$ must order one of two ways: either there exists an unconditional definition d' and the order is $c_1 \rightarrow^+ d' \rightarrow^+ c_2$, or the order is $c_2 \rightarrow^+ c_1$. $t_2(r)$ must order c_1 before c_2 and must order all unconditional definitions d' either before c_1 or after c_2. Vertex u is upwards exposed in the subtree rooted at c_2 in t_1; thus, by (P4), u is upwards exposed in the subtree rooted at c_2 in t_2. Since $d \rightarrow_{li} u$ is not in G, and u is upwards exposed in the subtree rooted at c_2 in t_2, there must be a definition d'' in the subtree rooted at c_1 that reaches u in t_2. This $d'' \rightarrow_{li} u$ is in G. If $t_1(r)$ orders unconditional definition d' in between c_1 and c_2, then $d'' \rightarrow_{li} u$ could not be in t_1's PDG. Therefore, t_1 would be a bto.

If, instead, t_1 orders by $c_2 \rightarrow^+ c_1$, then t_1 does not respect the edge $d'' \rightarrow_{li} u$ and must be a bto. Contradiction.

• $d \rightarrow_f u$ is loop carried. Since $d \rightarrow_{lc} u$ is not in G, there must exist an unconditional definition d' such that $t_1(r)$ orders by $c_1 \rightarrow^+ d'$ or $d' \rightarrow^+ c_2$. c_1 must follow c_2 in $t_2(r)$. Therefore, $t_2(r)$ must order all unconditional definitions d' by $c_2 \rightarrow^+ d' \rightarrow^+ c_1$.

u must be upwards exposed in the subtree rooted at c_2 in t_1 (because $d \rightarrow_{lc} u$ is in G') and in t_2 (by (P4)). Either u is upwards exposed in the subtree rooted at r in both t_1 and t_2, or there is an unconditional definition d', where d' is a child of r such that $d' \rightarrow_{li} u$. In the latter case, a contradiction arises immediately since t_2 orders all unconditional definitions after c_2. t_2 must be a bto since it does not preserve $d' \rightarrow_{li} u$.

Consider the former case, where u is upwards exposed in the subtree rooted at r in G. Suppose $t_1(r)$ orders by $d' \rightarrow^+ c_2$. d' kills the upwards exposedness of u in the subtree r, which implies that t_1 is a bto. Instead, suppose $t_1(r)$ orders by $c_1 \rightarrow^+ d'$. Because u is upwards exposed in the subtree rooted at r in G in t_1, there can be no unconditional definitions before c_2 in $t_1(r)$. Thus, either $t_1(r)$ orders by $c_2 \rightarrow^+ c_1 \rightarrow^+ d'$ or by $c_1 \rightarrow^+ c_2 \rightarrow^+ d'$. In both orderings, d' (or some unconditional definition following d') must reach u in t_1 (without loss of generality, assume d' is the one) by a loop-carried dependence. This implies that $d' \rightarrow_{lc} u$ is in t_2, which further implies that $d \rightarrow_{lc} u$ is in t_2, since $t_2(r)$ orders d' before c_1. Therefore, there must be a def-order edge $d' \rightarrow_{do} c_1$ in G. However, we assumed $t_1(r)$ orders by $c_1 \rightarrow^+ d'$, which could not preserve the def-order edge. Thus, t_1 is a bto. Contradiction.

4.2.2. Def-order edges identical

Consider def-order edge $d_0 \rightarrow_{do(u)} d_1$. There are five cases to consider:

(1) Both d_0 and d_1 are outside the subtree rooted at r.
(2) d_0 is inside the subtree rooted at r; d_1 is outside.
(3) d_1 is inside the subtree rooted at r; d_0 is outside.
(4) Both d_0 and d_1 are inside the subtree rooted at r. d_0 and d_1 are in the same subtree rooted at c, a control child of r.

For these four cases, the order of d_0 with respect to d_1 must be the same in both t_1 and t_3. Since the sets of loop-independent and loop-carried edges in the two graphs are identical $d_0 \rightarrow_{do} d_1$ must be in both G and G' if it is in either.

(5) Both d_0 and d_1 are inside the subtree rooted at r. d_0 and d_1 are in different subtrees (rooted at c_1 and

c_2, control children of r)

Assume that $d_0 \longrightarrow_{do} d_1$ is in G. Both t_1 and t_2 must order $c_1 \longrightarrow^+ c_2$. Since $t_3(r) = t_2(r)$, t_3 also orders $c_1 \longrightarrow^+ c_2$. Since loop-independent and loop-carried edges are identical in G and G', $d_0 \longrightarrow_{do} d_1$ must be in G'.

Assume that $d_0 \longrightarrow_{do} d_1$ is in G', but not in G. This implies that $t_3(r)$ orders $c_1 \longrightarrow^+ c_2$. Since $t_3(r) = t_2(r)$, t_2 must also order $c_1 \longrightarrow^+ c_2$. However, since both $d_0 \longrightarrow_f u$ and $d_1 \longrightarrow_f u$ are in G, there must be a def-order edge between d_0 and d_1 in G, and since t_2 orders $c_1 \longrightarrow^+ c_2$, the edge must run from d_0 to d_1.

References

Aho86.
Aho, A., Sethi, R., and Ullman, J., *Compilers: Principles, Techniques and Tools*, Addison-Wesley, Reading, MA (1986).

Ferrante87.
Ferrante, J., Ottenstein, K., and Warren, J., "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems* 9(5) pp. 319-349 (July 1987).

Horwitz87.
Horwitz, S., Prins, J., and Reps, T., "Integrating non-interfering versions of programs," Report 690, Department of Computer Sciences, University of Wisconsin—Madison (March, 1987).

Horwitz89.
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).