PDGs and Slicing

K. V. Raghavan

IISc

References

- Our primary reference: "The Semantics of Program Slicing". Thomas Reps and Wuu Yang. Technical Report, 1988. (*Available on course web page.*
- More detailed references:
  - PDGs, their construction, and their applications: "The program dependence graph and its use in optimization". J. Ferrante, K. J. Ottenstein, and J. D. Warren. 1987.
  - Semantics of PDGs: "On the adequacy of program dependence graphs for representing programs". S. Horwitz, J. Prins, and T. Reps. 1988.
  - Survey articles on different techniques for program slicing, and applications: (1) "A survey of program slicing techniques". F. Tip. 1995. (2) "A brief survey of program slicing". B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen. 2005.

The language under consideration

Language features that we consider

- Scalar variables only.
  - No pointers, arrays, structures, and dynamic memory allocation.
- Assignments; sequences of statements; “while” loops; “if then else” statements.
  - No gotos, breaks, continues, and ”return” statements. No exceptions (except terminating exceptions).
- Procedures.
  - Each procedure has parameters, and can potentially return a value.
  - Global variables.

The language under consideration

Language features that we consider

- Scalar variables only.
  - No pointers, arrays, structures, and dynamic memory allocation.
- Assignments; sequences of statements; "while" loops; "if then else" statements.
  - No gotos, breaks, continues, and "return" statements. No exceptions (except terminating exceptions).
- Procedures.
  - Each procedure has parameters, and can potentially return a value.
  - Global variables.

What to do about realistic languages?

- Researchers have shown how to extend PDGs and slicing to address these.
- However, semantics and proofs become harder.

The language under consideration

Language features that we consider

- Scalar variables only.
  - No pointers, arrays, structures, and dynamic memory allocation.
- Assignments; sequences of statements; “while” loops; “if then else” statements.
  - No gotos, breaks, continues, and ”return” statements. No exceptions (except terminating exceptions).
- Procedures.
  - Each procedure has parameters, and can potentially return a value.
  - Global variables.

What to do about realistic languages?

- Researchers have shown how to extend PDGs and slicing to address these.
- However, semantics and proofs become harder.

Initially, we restrict ourselves to single-procedure programs.

Program Dependence Graph (PDG)

- A program representation.
- Originally proposed by Ottenstein and Ottenstein in 1984. Fully described in their 1987 paper by Ferrante, Ottenstein, and Warren.
- Originally proposed applications
  - Slicing. [O and O, 1984]
  - Compiler optimizations, such as detection of parallelism, code motion, loop fusion, branch deletion, loop peeling and unrolling. [F, O, and W, 1987]

Overview

- Nodes in a PDG are nothing but nodes in CFG: assignments and predicates.
- Edges are of two kinds: control dependence and data dependence.
  - Data dependence edges, in turn, are of two kinds: flow dependences, and def-order dependences.

Overview

- Nodes in a PDG are nothing but nodes in CFG: assignments and predicates.
- Edges are of two kinds: control dependence and data dependence.
  - Data dependence edges, in turn, are of two kinds: flow dependences, and def-order dependences.

(This part of the lecture is based on Section 2.1 in our primary reference.)

Nodes in the PDG

Consider a (single-procedure) program P. Let G_P be its PDG. Nodes in G_P are:

- All assignments and predicates in P.
- A distinguished *entry* vertex.
- An *initial definition* vertex "$x := InitialState(x)$", for each variable x that is used in the program before being defined.
  - This vertex is treated as representing a (pseudo) assignment statement in the program at the beginning of the program.
- A *final use* vertex "$FinalUse(x)$" for each variable x whose final value is of interest to the user.
  - This vertex is treated as representing a (pseudo) assignment statement at the end of the program that reads x and writes to some dummy variable.

Control dependence edges

- We have a control-dependence edge $v_1 \rightarrow_c v_2$ if
  - v_1 is a predicate.
  - v_2 is encountered in all paths from one of the edges out of v_1 to the program exit, but not in all paths from the other edge out of v_1.
    - Paths are wrt the CFG.

  The edge is labeled *true* or *false*, depending on the edge out of v_1 along which v_2 is guaranteed to be encountered. Also, we say that v_2 is control-dependent on v_1.
- There is also a control-dependence edge (labeled *true*) from the entry vertex to every vertex that is present in all paths from the entry of the program to the exit of the program.

Properties of control dependence edges

- A node cannot be both *true* and *false* control-dependent on another node.
- For the language under consideration (i.e., when there are no jumps):
  - All edges going out of a "while" predicate are labeled *true*
  - Not counting *true* self-cycles from each "while" predicate to itself, the control-dependence edges induce a tree that is rooted at the entry vertex, and that mirrors the nesting structure of the program.

Flow dependences

A program dependence graph contains a flow dependence edge from vertex v_1 to vertex v_2 iff all of the following hold:

i) v_1 is a vertex that defines variable x.

ii) v_2 is a vertex that uses x.

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

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

(*This text, as well as several others that follow, copied from primary reference.*)

Two types of flow dependence edges

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

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

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

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

Need for making the distinction

Consider two different fragments:

v_1: while (..) {
v_2: sum = sum + x;
v_4: if (..)
v_3: x = x + 1;
}

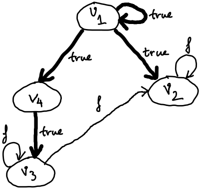
v_1: while (..) {
v_4: if (..)
v_3: x = x + 1;
v_2: sum = sum + x;
}

Need for making the distinction

Consider two different fragments:

v_1: while (..) {	v_1: while (..) {
v_2: sum = sum + x;	v_4: if (..)
v_4: if (..)	v_3: x = x + 1;
v_3: x = x + 1;	v_2: sum = sum + x;
}	}

- If there was no distinction, both fragments would yield the PDG:



- Two non-equivalent programs yield same PDG, which is bad!
- With distinction, first fragment yields $v_3 \rightarrow_{lc(L)} v_2$, where L is the loop in the fragment, while second fragment yields $v_3 \rightarrow_{li} v_2$ and $v_3 \rightarrow_{lc(L)} v_2$.

Def-order dependences

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

i) v_1 and v_2 both define the same variable.

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

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

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

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

v_3 is said to be the "witness" of the def-order edge.

Need for def-order dependences

Consider two different fragments:

v_1: if (p)
v_2: x = 1;
v_3: if (q)
v_4: x = 2;
v_5: y = x + 3;

v_3: if (q)
v_4: x = 2;
v_1: if (p)
v_2: x = 1;
v_5: y = x + 3;

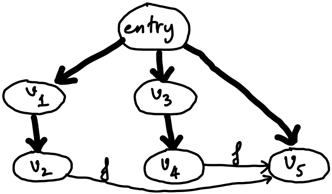
Need for def-order dependences

Consider two different fragments:

v_1: if (p)
v_2: x = 1;
v_3: if (q)
v_4: x = 2;
v_5: y = x + 3;

v_3: if (q)
v_4: x = 2;
v_1: if (p)
v_2: x = 1;
v_5: y = x + 3;

- If there were no def-order edges, both fragments would yield the PDG fragment:



  - Two non-equivalent programs yield same PDG, which is bad!
- Otherwise, we get the edge $v_2 \rightarrow_{do(v_5)} v_4$ with the first fragment and $v_4 \rightarrow_{do(v_5)} v_2$ with the second fragment.

A PDG is a multi-graph

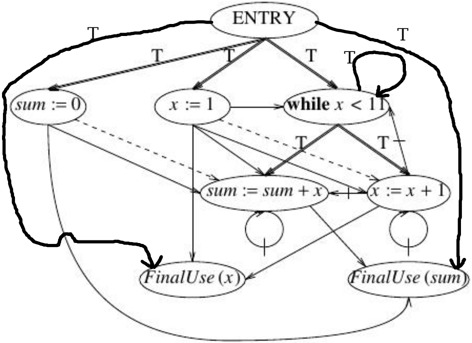
- From a node v_i to a node v_j, there could be multiple loop-carried edges, each one carried by a different loop.
- From a node v_i to a node v_j, there could be multiple def-order edges, each one having a different witness.

Example PDG

```
program Main
    sum := 0;
    x := 1;
    while x < 11 do
        sum := sum + x;
        x := x + 1
    od
end(x, sum)
```

Example PDG

```
program Main
  sum := 0;
  x := 1;
  while x < 11 do
    sum := sum + x ;
    x := x + 1
  od
end(x, sum)
```



- Bold arrows represent control dependence edges, dashed arrows represent def-order dependence edges, solid arrows represent loop-independent flow dependence edges, and solid arrows with a hash mark represent loop-carried flow dependence edges.

Definition: sequence of values at a node

Consider a run of a program P on an initial state such that the program halts.

- At any point of time in the run, if execution is at a program point v, the *value at v* at that time point is defined to be
  - the value assigned to the lhs if v is an assignment statement
  - the boolean result if v is a predicate
  - the value in variable x if v is "*FinalUse(x)*"
- the *sequence of values computed by P at a program point v* is the (finite) sequence of values at v across the entire run.

Adequacy of PDGs

- PDGs are an abstract program representation. That is, in general they contain less information than a program's text or its CFG.
- However, they are adequate to represent the semantics of a program.
  - That is, two programs with isomorphic PDGs are equivalent.

Definition of PDG isomorphism

G_P and G_Q are isomorphic iff there exists a bijective function from $V(G_P)$ to $V(G_Q)$ such that:

- Each pair of mapped nodes have internal expressions of the same structure. That is, corresponding operators and constants must match. (Corresponding variable names need not be the same.)
- An edge $v_1 \rightarrow v_2$ exists in G_P iff an edge exists from v_1' to v_2' in G_Q, such that:
  - Both edges are of the same kind (control/flow/def-order).
  - The edge labels (i.e., *true/false/li/lc)* match.
  - If the two edges are *lc*, then the carrying loop's headers are mapped under the bijection.
  - If the two edges are def-order, then the witnesses are mapped under the bijection.
  - If the two edges are flow dependence edges, they flow into corresponding operand positions of v_2/v_2'.

  where v_1' is the vertex that v_1 is mapped to and v_2' is the vertex that v_2 is mapped to under the bijection.

Adequacy of PDGs – formal statement

Theorem in Section 4.2.2:

Suppose that P and Q are programs for which $G_P \approx G_Q$ (i.e., G_P and G_Q are isomorphic). If σ is a initial state on which P halts, then for any state σ' that agrees with σ on all variables for which there are initial-definition vertices in P : (1) Q halts on σ', (2) P and Q compute the same sequences of values at corresponding program points, and (3) the final states agree on all variables for which there are final-use vertices in G_P.

Adequacy of PDGs – formal statement

Theorem in Section 4.2.2:
Suppose that P and Q are programs for which $G_P \approx G_Q$ (i.e., G_P and G_Q are isomorphic). If σ is a initial state on which P halts, then for any state σ' that agrees with σ on all variables for which there are initial-definition vertices in P : (1) Q halts on σ', (2) P and Q compute the same sequences of values at corresponding program points, and (3) the final states agree on all variables for which there are final-use vertices in G_P.

Notes:

- It is possible for two non-identical programs P and Q (i.e., with non-isomorphic CFGs) to have isomorphic PDGs.
- In this case, consider runs of P and Q on agreeing initial states σ and σ'. Also, consider two corresponding instances of any node v in these two runs:
  - The values at v in these two instances are guaranteed to be equal.
  - However, the entire memory states at these two instances may not match.

Illustration of PDG isomorphism and program equivalence

```
c = InitialState(c)
i = 0;
j = 0;
while (i < 100) {
  i = i + 2; // values: 2, 4, 6,
  j = j - 2; // values: -2, -4, -6
  c = c + i + j;
}
finaluse(c)
```

```
c = InitialState(c)
i = 0;
j = 0;
while (i < 100) {
  j = j - 2;   // -2 -4, -6
  i = i  + 2; // 2, 4, 6
  c = c + i + j;
}
finaluse(c)
```

What is a slice?

There are many different definitions of a slice in the literature. What follows in one commonly used definition. (This definition is not available in the paper.)

Let P be a program, and S be a *criterion*, namely, a subset of statements and predicates in the program. A program Q is said to be a (correct) slice of P wrt to S if

- Q consists of some subset of the nodes in P.
- Q includes all nodes in S.
- The initial definition nodes in Q are a subset of the initial definition nodes in P, and every variable that is used before being defined in Q has an initial definition node in Q.
- For any state σ on which P halts, and for any state σ' that agrees with σ on all variables for which there are initial-definition vertices in Q: (1) Q halts on σ', and (2) For each node v in Q, P computes the same sequence of values at its copy of v as Q does at v.

Approach described in our primary reference to compute a slice

(This part of lecture derived from Section 2.2.)

- For a vertex s of a PDG G, the operation "/", discussed below, produces a PDG G/s which is a slice of G wrt s.
  G/s contains all vertices in G on which s has a transitive flow or control dependence (i.e. all vertices that can reach s via flow or control edges).
  That is, $V(G/s) = \{w \mid w \in V(G), w \rightarrow^*_{c,f} s\}$.
  (Here, by flow edges, we mean both kinds of flow edges.)
- The approach is extended to the setting where the criterion is a set of vertices S as follows:

$$V(G/S) = \bigcup_{s \in S} V(G/s)$$

- For any $v \notin G$, $V(G/v)$ is defined as $\emptyset$.

Approach – contd.

The edges in the graph G/S are essentially those in the subgraph of G *induced* by $V(G/S)$, with the exception that a def-order edge $v \rightarrow_{do(u)} w$ is only included if, in addition to v and w, $V(G/S)$ also contains the vertex u.

Feasibility of a slice

(This part of the lecture is based on Section 3.)
LEMMA (FEASIBILITY): For any program P and subset S of nodes in G_P, $G_Q \equiv (G_P/S)$ is a *feasible* PDG. That is, there exists a program Q whose PDG is isomorphic to G_Q.

Informal Proof of Feasibility Lemma

- The proof is by showing a technique to construct a sliced program Q by *projecting out* nodes from P, as follows. Initially, set Q be equal to P itself. Then, from Q remove
  - each assignment statement whose node is not present in G_P/S
  - each "if" or "while" block whose predicate is not present in G_P/S
    - It is guaranteed that no node inside the block will be included in G_P/S.
- It can be shown that the PDG of the program Q produced above is isomorphic to G_P/S. (See proof in Section 3.)

Informal Proof of Feasibility Lemma

- The proof is by showing a technique to construct a sliced program Q by *projecting out* nodes from P, as follows. Initially, set Q be equal to P itself. Then, from Q remove
  - each assignment statement whose node is not present in G_P/S
  - each "if" or "while" block whose predicate is not present in G_P/S
    - It is guaranteed that no node inside the block will be included in G_P/S.
- It can be shown that the PDG of the program Q produced above is isomorphic to G_P/S. (See proof in Section 3.)
- A note about the construction above: Informally speaking, the relative ordering of statements in Q is guaranteed to be the same as that in P. Therefore, the approach works even if we exclude all def-order edges from G_P (and hence, from G_Q).

Another way to construct a sliced program

- Say we have G_P and G_P/S, but don't have access to (the CFG of) P.
- In this setting, we would need to include def-order edges in G_P and in G_P/S.
- A naive approach to construct Q:
  - Enumerate by brute-force programs that have the same nesting structure (i.e., the same control-dependence subgraph of the PDG) as P, until a program is found whose PDG is isomorphic to G_P/S.

Correctness

- Theorem 1: Any program whose PDG is isomorphic to $G_Q \equiv (G_P/S)$ is a correct slice of P wrt S. (Proof in Section 4.)

Correctness

- Theorem 1: Any program whose PDG is isomorphic to $G_Q \equiv (G_P/S)$ is a correct slice of P wrt S. (Proof in Section 4.)
- G_Q in fact satisfies a stronger property, as follows (this property is not mentioned in the paper). Say we construct a program Q' by *transforming* certain nodes in P as follows:
  - Replace each assignment statement $v \equiv$ "$x = expr$" in P that is not in G_Q with "$x = *$", where "$*$" is a non-deterministically chosen value.
  - Replace each predicate p in P that is not in G_Q with "$*$", where "$*$" is a non-deterministically chosen boolean value.

  Then, Q' generates the same sequence of values as P does at all nodes that were *not* transformed as mentioned above, when P and Q' are run on agreeing initial states σ and σ' such that both runs terminate normally. (Note: Q' is a transformed version of P. Q' is not a slice of P wrt S.)

Notes about the strong property

- Let Q be a correct slice of P wrt S such that Q's PDG is not isomorphic to G_P/S (i.e., Q was not constructed by the PDG-based approach presented above).
- If we produce Q' by transforming nodes in P (as discussed in the previous slide) that are not in Q, then such a Q' *may not* satisfy the strong property.
- In other words, the strong property is not satisfied by arbitrary slices. It is satisfied only by slices produced by the PDG-based approach presented above.

An application of the strong property in the context of debugging

- Say during a run of a program P (using a test input) we are getting an unexpected value at some instance of some node v (e.g., a printf node).
- As per the strong property, the bug *cannot* be fixed by modifying any node of P that is not in G_P/v. This is because Q' (constructed as mentioned earlier, using G_P/v) gives the same unexpected value for the same test input as does P.

Other applications of PDGs/slicing

- Classifying changes to a program (between two versions of the program) as textual changes vs. semantic changes.
- Merging two different variants of a base version of a program.
- Identifying duplicated code fragments.
- Program testing
  - Selecting a subset of test cases (from a test suite) that still give high coverage.
  - Selecting a subset of test cases (from a test suite) to cover recently modified statements.
- To reduce the size of a program in order to analyze it more efficiently (when a criterion is known).

Other techniques to compute slices

- [Weiser 1981] is the original technique. It is more expensive, and no more precise than the PDG-based technique.
- There are many subsequent techniques that are more precise than the PDG-based technique.
  - They usually compute a "path sensitive" slice.
- There can be no technique that always computes the most-precise slice.

Other kinds of slices

The kind of slice that we discussed so far was a static, syntactic, backward slice.

Other kinds of slices:

- A dynamic (as opposed to *static*) slice of P wrt S is a slice that is correct only wrt to a given initial state σ.
  - Useful during debugging, and during dynamic analysis (i.e., analysis of a program restricting attention to a specific run).
- A semantic (as opposed to *syntactic*) slice is a program Q that is not necessarily a projection of the given program P. It could be a arbitrarily transformed version of P. The guarantees are that (1) the nodes in S are present in Q, and (2) the same sequence of values is computed at the nodes in S by P and by Q starting from initial states σ and σ' that agree on variables that have initial-definitions in P.
- A forward (as opposed to *backward*) slice includes nodes in P that depend on S, and not vice versa. The semantic properties of forward slices are different from those of backward slices.