A Program Slicing Algorithm

Phillip James
Swansea University

May 19, 2009

Today's Talk

- Introduction to Program Slicing.
- Dependencies and Graphs.
- An Algorithm for Program Slicing.
- Correctness of the Algorithm.
- Towards Slicing in Verification.

Introduction to Program Slicing

Program Slicing Intuitively

Given an imperative program, a slice is an executable program whose behaviour must be identical to the specified subset of the original program's behaviour. Mark Weiser (1984).

- Weiser observed how people debug a program.
- Slicing applications include:
  - program debugging,
  - program maintenance,
  - and program verification.

Example of Slicing

Slicing Criterion: Point 10

```
1  public void Calc(n){
2    i = 1;
3    prod = 1;
4    sum = 0;
5    while (i<=n){
6      sum = sum + 1;
7      prod = prod * i;
8      i++;
9    }
10   Print(sum);
11   Print(prod);
12 }
```

Original Program

Example of Slicing

Slicing Criterion: Point 10

```
1  public void Calc(n){
2     i = 1;
3     prod = 1;
4     sum = 0;
5     while (i<=n){
6        sum = sum + 1;
7        prod = prod * i;
8        i++;
9     }
10    Print(sum);
11    Print(prod);
12 }
```

Original Program

```
1  public void Calc(n){
2     i = 1;
3
4     sum = 0;
5     while (i<=n){
6        sum = sum + 1;
7
8        i++;
9     }
10    Print(sum);
11
12 }
```

Slice

Dependencies and Graphs

Program Dependencies

There are two main dependencies relied upon in program slicing:

1. Control Flow Dependencies (e.g. Branches) represented using Control Flow Graphs.
2. Data Dependencies (e.g. $x := y$) represented using Program Dependence Graphs.

Definitions given follow those of Ball and Horwitz: Slicing Programs with Arbitrary Control Flow, LNCS, 1993.

Control Flow Graphs (CFG) – Definition

Definition: Control Flow Graph

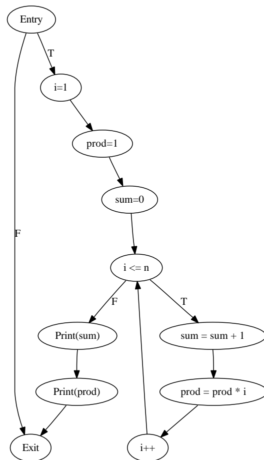
A Control Flow Graph is a labelled, directed, rooted graph with the following conditions:

- Three types of vertices: Fall through (one successor), predicate (two successors), and an *EXIT* vertex (no successors).
- The root of the graph is the *ENTRY* vertex (a predicate vertex).
- *EXIT* is reachable from all vertices.
- Edges are labelled with `true` or `false` when leaving a predicate vertex, and `null` otherwise.

Example Control Flow Graph

```
 1 public void Calc(n){
 2   i = 1;
 3   prod = 1;
 4   sum = 0;
 5   while (i<=n){
 6     sum = sum + 1;
 7     prod = prod * i;
 8     i++;
 9   }
10   Print(sum);
11   Print(prod);
12 }
```

Note: Translation, program to CFG and back is easy.



Imperative Language Under Consideration

- Expressions only contain scalar variables and constants.
- Statements are only:
  - assignments,
  - outputs,
  - jumps,
  - conditionals,
  - or loops.

Operational Semantics of a Control Flow Graph

Let σ represent the state, i.e. a mapping from variables to values.

Operational Semantics of a Control Flow Graph

Let σ represent the state, i.e. a mapping from variables to values. For a CFG G the operational semantics $G(\sigma)$ is given by a (possibly infinite) state sequence recording the state at every passed vertex:

Operational Semantics of a Control Flow Graph

Let σ represent the state, i.e. a mapping from variables to values. For a CFG G the operational semantics $G(\sigma)$ is given by a (possibly infinite) state sequence recording the state at every passed vertex:

- Start Execution at *Entry*, with initial state σ.

Operational Semantics of a Control Flow Graph

Let σ represent the state, i.e. a mapping from variables to values. For a CFG G the operational semantics $G(\sigma)$ is given by a (possibly infinite) state sequence recording the state at every passed vertex:

- Start Execution at *Entry*, with initial state σ.
- At any stage, there is a single point of control with a state mapping variables.

Operational Semantics of a Control Flow Graph

Let σ represent the state, i.e. a mapping from variables to values. For a CFG G the operational semantics $G(\sigma)$ is given by a (possibly infinite) state sequence recording the state at every passed vertex:

- Start Execution at *Entry*, with initial state σ.
- At any stage, there is a single point of control with a state mapping variables.
- Executing a flow through vertex, performs the action on the label of the vertex (possibly changing the state), and passes control to its successor.

Operational Semantics of a Control Flow Graph

Let σ represent the state, i.e. a mapping from variables to values. For a CFG G the operational semantics $G(\sigma)$ is given by a (possibly infinite) state sequence recording the state at every passed vertex:

- Start Execution at *Entry*, with initial state σ.
- At any stage, there is a single point of control with a state mapping variables.
- Executing a flow through vertex, performs the action on the label of the vertex (possibly changing the state), and passes control to its successor.
- Executing a predicate vertex evaluates the conditional and then passes control to a single successor depending on the result.

Operational Semantics of a Control Flow Graph

Let σ represent the state, i.e. a mapping from variables to values. For a CFG G the operational semantics $G(\sigma)$ is given by a (possibly infinite) state sequence recording the state at every passed vertex:

- Start Execution at *Entry*, with initial state σ.
- At any stage, there is a single point of control with a state mapping variables.
- Executing a flow through vertex, performs the action on the label of the vertex (possibly changing the state), and passes control to its successor.
- Executing a predicate vertex evaluates the conditional and then passes control to a single successor depending on the result.
- execution terminates if *EXIT* is reached.

Characterising Vertex Behaviour

Definition: Vertex Behaviour

For an execution $G(\sigma)$, the behaviour of a vertex is characterised by the sequence of values that arise from the vertex. If the vertex is:

- Assignment statement – Values assigned to variable.
- Output statement – Values output.
- Predicate vertex – Boolean values to which predicate evaluates.

The values arising at a vertex v are denoted by $G(\sigma)(v)$.

Equivalent Vertices

Definition: Equivalence of Vertices

Let G and H be CFG's and let v_G, v_H be vertices of G and H respectively.

v_G is equivalent to v_H iff, for all σ we have:

- $G(\sigma)\text{term} \wedge H(\sigma)\text{term} \wedge G(\sigma)(v_G) = H(\sigma)(v_H)$ or
- $\neg G(\sigma)\text{term} \wedge \neg H(\sigma)\text{term} \wedge (G(\sigma)(v_G) \sqsubseteq H(\sigma)(v_H) \vee H(\sigma)(v_H) \sqsubseteq G(\sigma)(v_G))$ or
- $G(\sigma)\text{term} \wedge \neg H(\sigma)\text{term} \wedge H(\sigma)(v_H) \sqsubseteq G(\sigma)(v_G)$ or
- $\neg G(\sigma)\text{term} \wedge H(\sigma)\text{term} \wedge G(\sigma)(v_G) \sqsubseteq H(\sigma)(v_H)$.

Where term = terminates, $\sqsubseteq$ = "is a prefix of".

Program Dependant Graphs – Some Preliminaries

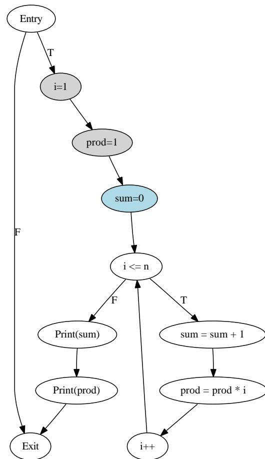
Definition: Postdominance

Let v and w be vertices in a CFG. Vertex w postdominates vertex v iff $w \neq v$ and w is on every path from v to the *EXIT* vertex.

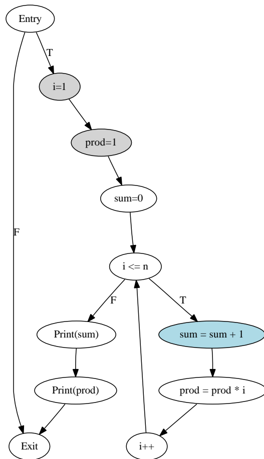
Lemma:

It is impossible for vertex v to postdominate vertex w and vertex w to postdominate vertex v.

Postdominance Example



Non-Postdominance Example

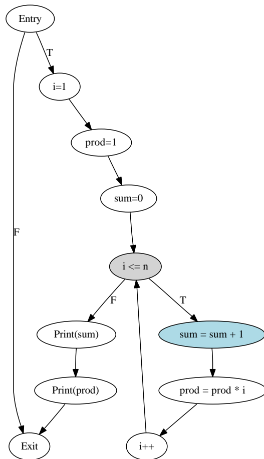


Program Dependant Graphs – Some Preliminaries

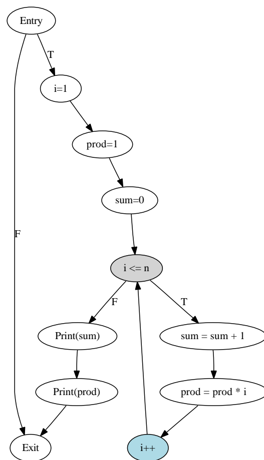
Definition: L-Postdominance

w postdominates the L-branch (L = true or false) of predicate vertex v iff w is the L-successor of v or w postdominates the L-successor of v.

T-Postdominance Example (immediate successor)



T-Postdominance Example



Program Dependant Graphs – Some Preliminaries

Definition: L-Control Dependence

Let v and w be vertices in a CFG. Vertex w is directly L-control dependant on v iff w postdominates the L-branch of v and w does not postdominate v.

Program Dependant Graphs – Some Preliminaries

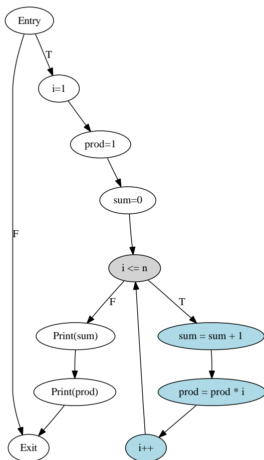
Definition: L-Control Dependence

Let v and w be vertices in a CFG. Vertex w is directly L-control dependant on v iff w postdominates the L-branch of v and w does not postdominate v.

Notation: Control Dependence

v is control dependant on w if v is T-control dependant on w or v is F-control dependant on v.

T-Control Dependence Example



Program Dependence Graphs – Some Preliminaries

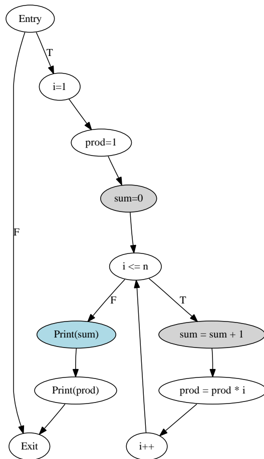
Definition: Flow Dependence

Let v and w be vertices in a CFG.

There is a flow dependence from v to w iff,

- vertex v assigns to a variable x,
- vertex w uses x and
- there is a path from v to w that does not include an assignment to x.

Flow Dependence Example



Program Dependence Graphs – Definition

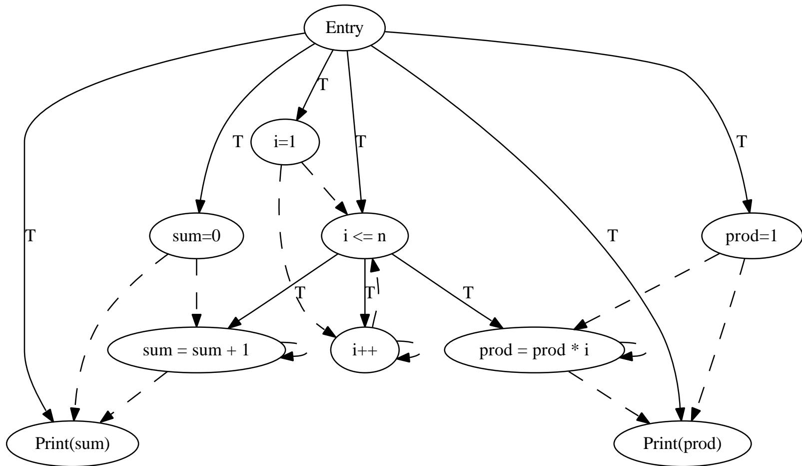
A program dependence graph (PDG) is defined in terms of a programs control flow graph.

Definition: Program Dependence Graph

For a CFG G, we construct the program dependence graph using following rules,

- $V(PDG) = V(CFG) - \{EXIT\}$.
- $n \rightarrow_c m$ iff m is control dependant on n.
- $n \rightarrow_f m$ iff m is flow dependant on n.

Example Program Dependence Graph



$FlowDependant(- \rightarrow)$ $ControlDependant(\longrightarrow)$

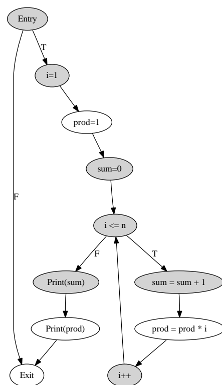
An Algorithm for Program Slicing

Ball and Horwitz Algorithm '92

```
1.  Function Slice (P:program, c:slicing criterion)
2.  declare
3.    v: vertex, G: control flow graph, D: program dependence graph
4.    S: subset of D's vertices, Q: projection of P
5.  begin
6.   /*Step 0: Build The PDG*/
7.     G := the control flow graph for P
8.     D := the program dependency graph that corresponds to G
9.   /*Step 1: Identify Vertices*/
10.    v := vertex in G corresponding to criterion c.
11.    S := {w | v is reachable from vertex w in D} + {EXIT}
12.  /*Step 2: Create The Program Projection*/
13.    Q := P
14.    eliminate from Q all stmt subtrees T such that
15.    vert(T) are not in S
16.    return(Q)
17. end
```

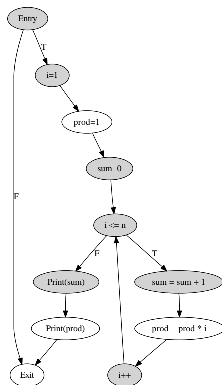
Algorithm Example

Vertices identified by `Step 1`, with slicing criterion “Print(sum)”:



Algorithm Example

Vertices identified by `Step 1`, with slicing criterion “Print(sum)”:




```
1  public void Calc(n){
2    i = 1;
3
4    sum = 0;
5    while (i<=n){
6      sum = sum + 1;
7
8      i++;
9    }
10   Print(sum);
11
12 }
```

Slice

Correctness – Some Preliminaries

Definition: Path-projection

Graph H is a path-projection of graph G iff $V(H) \subseteq V(G)$ and,

- For every path in G, if the vertices that are not in $V(H)$ are eliminated along their outgoing edge labels, then the resulting sequence is a path in H.
- For every path PT in H, there exists a path in G whose projection is PT.

Definition: Flow/path-projection

Given a control flow graph G, a control flow graph H is a flow/path-projection of G iff, H is a path-projection of G and:

for every vertex w in $V(H)$, if $PDG(G)$ contains $v \rightarrow_f w$, then v is also in $V(H)$.

Correctness Proof

Theorem: Partial Correctness of Slice

The algorithm "Slice" is partially correct, i.e. for a program P and a slicing criterion C, the algorithm produces a projection H of $CFG(P)$ such that both $CFG(P)$ and H have equivalent behaviour at every shared component, including all vertices of $CFG(P)$ corresponding to C.

Proof of termination of the algorithm is trivial since all program graphs are finite.

Correctness Proof – Abstract

The correctness proof can be split into 3 main parts:

1. Theorem 1 – Regards CFG's. If CFG H is a flow/path-projection of CFG G then for all vertices v in $V(H)$, v in G and v in H are equivalent, i.e., have the same behaviour.

Correctness Proof – Abstract

The correctness proof can be split into 3 main parts:

1. Theorem 1 – Regards CFG's. If CFG H is a flow/path-projection of CFG G then for all vertices v in $V(H)$, v in G and v in H are equivalent, i.e., have the same behaviour.
2. Theorem 2 – The algorithm constructs a flow/path projection. Given a CFG G, Step 1 produces a set of vertices S, such that eliminating vertices not in S from G produces a CFG H that is a flow/path-projection of G.

Correctness Proof – Abstract

The correctness proof can be split into 3 main parts:

1. Theorem 1 – Regards CFG's. If CFG H is a flow/path-projection of CFG G then for all vertices v in $V(H)$, v in G and v in H are equivalent, i.e., have the same behaviour.
2. Theorem 2 – The algorithm constructs a flow/path projection. Given a CFG G, Step 1 produces a set of vertices S, such that eliminating vertices not in S from G produces a CFG H that is a flow/path-projection of G.
3. Theorem 3 – The algorithm constructs a program P' from a flow/path-projection. Given P, $CFG(P)$ and a flow/path-projection H of $CFG(P)$, produce a program P' such that $CFG(P') = H$.

Correctness Proof – Abstract

The correctness proof can be split into 3 main parts:

1. Theorem 1 – Regards CFG's. If CFG H is a flow/path-projection of CFG G then for all vertices v in $V(H)$, v in G and v in H are equivalent, i.e., have the same behaviour.
2. Theorem 2 – The algorithm constructs a flow/path projection. Given a CFG G, Step 1 produces a set of vertices S, such that eliminating vertices not in S from G produces a CFG H that is a flow/path-projection of G.
3. Theorem 3 – The algorithm constructs a program P' from a flow/path-projection. Given P, $CFG(P)$ and a flow/path-projection H of $CFG(P)$, produce a program P' such that $CFG(P') = H$.

Vertex Elimination

Definition: Vertex Elimination

Let G be a CFG. Let v be a vertex in $V(G)$.
Then $G - v$ denotes the CFG after eliminating vertex v, i.e.,

- $V(G - v) = V(G) \setminus \{v\}$.
- $E(G - v) = (E(G) \setminus Remove(G, v)) \cup New(G, v)$.
- $Remove(G, v) = \{(x, v) \in E(G)\} \cup \{(v, y) \in E(G)\}$.
- $New(G, v) = \{(x, y) | (x, v) \in E(G), (v, y) \in E(G)\}$
- $Label(x, y) = Label(x, v)$ for $(x, y) \in New(G, v)$ and (x, v) "generates" (x, y).

Correctness Proof

Construction of H:

Let S be the set of nodes produced in Step 1.
H is the graph resulting from G by removing all vertices not in S.

Lemma 1:

Let G be a CFG. Let $v, w \in V(G)$:
$(G - v) - w = (G - w) - v$.

Correctness Proof

Lemma 2:

The construction of S and H yields that:
for every vertex w in $V(H)$, if $PDG(G)$ contains $v \longrightarrow_f w$, then v is also in $V(H)$.

Correctness Proof

Lemma 2:

The construction of S and H yields that:
for every vertex w in $V(H)$, if $PDG(G)$ contains $v \longrightarrow_f w$, then v is also in $V(H)$.

Theorem 4:

Let G be a CFG and let S be a set of vertices such that if $w \in S$ and $v \longrightarrow_c w$ in the $PDG(G)$ then $v \in S$.

Eliminating the vertices not in S from G's CFG yields H which is a CFG and a path projection of G.

Correctness Proof

Lemma 3:

Let G be a CFG. Let $v \in V(G)$ such that $v \notin \{ENTRY, EXIT\}$:

- $G - v$ has $ENTRY$ as root.
- in $G - v$ $EXIT$ is reachable from all vertices.

Correctness Proof

Observation:

A single application of vertex elimination can produce a graph with a vertex that has two distinct L-successors.

Correctness Proof

Lemma 4:

H has no vertex v with two distinct L-successors.

Applying Slicing to Verification

Ladder Logic as a Program

- Ladder logic - language used to describe railway interlockings.
- Ladder logic program consists of 3 main steps inside a **while** statement.
  1. Read inputs.
  2. Perform assignments given by ladder logic.
  3. Output results.
- Having such a structure allows us to apply program slicing.

Verifying Safety Conditions

Given a particular safety condition ϕ, we want to check ϕ holds in all possible program states.

Problem: Ladder logic programs have a huge state space.

The state space can be reduced by slicing the program with regards to ϕ.

Limitations

- Fokkink and Hollingshead (1998) use slicing on ladder logic programs.
    - Algorithm published but no correctness proof.
- Ball and Horwitz. (1993)
    - Provide algorithm and correctness proof.
    - Excludes coverage of input statements.
- Hatcliff et. al. (2005)
    - Give algorithm to cover full Java.
    - Proof of correctness appears to miss some aspects.
- Other approaches usually without correctness proof.

Summary

To Summarise

- Overview of (static) slicing, including definition and examples.
- Structures involved in slicing (CFG's and PDG's).
- An algorithm to perform slicing.
- Shown the correctness of the algorithm.
- Looked at how program slicing can be used within verification.

References

Thomas Ball and Susan Horwitz.
Slicing programs with arbitrary control-flow.
In *AADEBUG '93: Proceedings of the First International Workshop on Automated and Algorithmic Debugging*, London, UK, 1993. Springer-Verlag.

Wan Fokkink and Paul Hollingshead.
Verification of interlockings: from control tables to ladder logic diagrams.
In *Proceedings of the 3rd Workshop on Formal Methods for Industrial Critical Systems - FMICS'98*, 1998.

Venkatesh Prasad Ranganath, Torben Amtoft, Anindya Banerjee, John Hatcliff, and Matthew B. Dwyer.
A new foundation for control dependence and slicing for modern program structures.
ACM Trans. Program. Lang. Syst., 29(5):27, 2007.

Frank Tip.
A survey of program slicing techniques.
Journal of Programming Languages, 3:121–189, 1995.

A Sample Rule For CFG Translation

```
stmt:
  NullStmt(){
    stmt.entry = stmt.cont
  }
| While(expr seq){
    stmt.entry = Pred("expr",seq.entry,stmt.cont)
    seq.cont = stmt.entry
  }
| ...
```