Interprocedural Slicing Using Dependence Graphs

Susan Horwitz, Thomas Reps, and David Binkley
Presented by Daniel DeFreeze

ECS 289C, Winter 2015
The slice of a program with respect to program point $p$ and variable $x$ consists of all statements and predicates of the program that might affect the value of $x$ at point $p$.

- Backward slicing
- Forward slicing
- Chop
program Main
  sum := 0
  i := 1
  while i < 11 do
    sum := sum + i
    i := i + 1
  od
end

Slice at line 6, variable \( i \)

program Main
  i := 1
  while i < 11 do
    i := i + 1
  od
end
Slicing

Why?

- Debugging
- Parallelization
- Pre-processing for further analysis
- ...

Program Dependence Graph

Used for intraprocedural slicing
The PDG for program $P$ is a directed graph with the following Vertices:

1. Assignment statements
2. Control predicates
3. Entry vertex
4. Other special cases, see paper

Edges:

1. Control Dependence
2. Data Dependence
CFG vs. Control Dependence

- Differs from Control Flow Graph (CFG)
- CFG is unnecessarily imperative
PDG Slice

(from backward reachability in PDG)
(procedures are call by value-result)
Interprocedural Slicing

1 program Main
2 \text{i} := 1
3 while \text{i} < 11 do
4 \text{call} A(\text{sum}, \text{i})
5 \text{od}
6 end

1 \text{procedure} A(x, y)
2 \text{call} \text{Add}(x, y);
3 \text{call} \text{Inc}(y);
4 \text{return}

1 \text{procedure} \text{Add}(a, b)
2 a := a + b
3 \text{return}

1 \text{procedure} \text{Inc}(z)
2 \text{call} \text{Add}(z, 1)
3 \text{return} \quad \text{// slice respect to z}

(procedures are call by value-result)
Interprocedural Slicing

(programs are call by value-result)
Contributions

- Introduce a new kind of graph - system dependence graph
- Show how to track transitive data dependences across procedures
- Give algorithm for interprocedural slicing over SDG
Overview of Steps

1. Construct program dependence graph for each procedure
2. Add call edges from call-site vertices to procedure-entry vertices
3. Add parameter-in edges from actual-in to formal-in vertices
4. Add parameter-out edges from formal-out to actual-out vertices
5. Construct linkage grammar
6. Compute subordinate characteristic graphs of linkage grammar nonterminals
7. Add transitive flow dependence edges
System Dependence Graph

- Program Dependence Graph for main procedure
- Procedure Dependence Graphs for each procedure
- New edges between call sites and procedure entries
- New edges for transitive dependencies due to calls
Procedure Dependence Graphs

- Vertex for procedure entry point
- Entry point is control dependent on call site
- Formal in, formal out, actual in, actual out vertices
- Parameter vertices control dependent on entry point
- Formal in data dependent on actual in, and actual out data dependent on formal out
Procedure Dependence Graphs

1 program Main
2    sum := 0
3    i := 1
4    while i < 11 do
5       call A(sum, i)
6    od
7 end
Procedure Dependence Graphs

1 program Main
2 sum := 0
3 i := 1
4 while i < 11 do
5 call A(sum, i)
6 od
7 end
program Main
sum := 0
i := 1
while i < 11 do
    call A(sum, i)
od
end
program Main
sum := 0
i := 1
while i < 11 do
    call A(sum, i)
od
end
Procedure Dependence Graphs

```plaintext
1 program Main
2 sum := 0
3 i := 1
4 while i < 11 do
5    call A(sum, i)
6  od
7 end
```
Procedure Dependence Graphs

```plaintext
ENTER main

i := 1

while i < 11

call A

i := y_out

y_in := i

sum := x_out

x_in := sum

sum := 0

FinalUse(i)

FinalUse(sum)
```
Procedure Dependence Graphs
System Dependence Graph

- Stitch together individual procedure dependence graphs
- See paper for full example
Overview of Steps

☑ Construct program dependence graph for each procedure
☑ Add call edges from call-site vertices to procedure-entry vertices
☑ Add parameter-in edges from actual-in to formal-in vertices
☑ Add parameter-out edges from formal-out to actual-out vertices

5 Construct linkage grammar
6 Compute subordinate characteristic graphs of linkage grammar nonterminals
7 Add transitive flow dependence edges
Attribute Grammars

- dependence graph between attributes
- Linkage grammar is generated so that attribute dependences model flows between procedures
- We don’t care about the language the grammar accepts, or the actual values of the attributes
Subordinate characteristic graphs
Nonterminal for each procedure Add occurrence of nonterminal on RHS for each call to procedure

\[
\begin{align*}
\text{Main} & \rightarrow A \\
A & \rightarrow \text{Add Inc} \\
\text{Add} & \rightarrow \varepsilon \\
\text{Inc} & \rightarrow Add
\end{align*}
\]
Attributes

- Attribute dependences are result of intraprocedural slicing.
- Procedures are sliced with respect to parameter vertices to procedure calls.
- Dummy attribute functions added to model parameter dependence.

```
program Main
  sum := 0
  i := 1
  while i < 11 do
    call A(sum, i)
  od
end

A.x_in = f1(A.x_out, A.y_out)
A.y_in = f2(A.y_out)
A.x_out = f3(A.y_out)
A.y_out = f4(A.y_out)
```
• Need to compute transitive attribute dependences
• Termed the **subordinate characteristic graph**
• See paper for examples of attribute dependences in linkage grammar
Overview of Steps

- Construct program dependence graph for each procedure
- Add call edges from call-site vertices to procedure-entry vertices
- Add parameter-in edges from actual-in to formal-in vertices
- Add parameter-out edges from formal-out to actual-out vertices
- Construct linkage grammar
- Compute subordinate characteristic graphs of linkage grammar nonterminals
- Add transitive flow dependence edges
Interprocedural Slicing with SDG

Two phases

1. Phase 1 does not follow parameter out edges (no procedures called by P).
2. Phase 2 does not follow call edges or parameter in edges.
Is it slow?

- Building SDG: $O(N^4)$ with respect to size of input, particularly number of procedure calls
- Slicing is fast after SDG is built
Questions?
Attribute Grammars

- Originally described by Knuth
- Attributes are attached to terminal and nonterminal symbols
- Attribute equations define attribute values for each attribute in a production, defined in terms of other attributes

\[
\begin{align*}
B & \rightarrow 0 \quad v(B) = 0 \\
B & \rightarrow 1 \quad v(B) = 2^{s(B)} \\
L & \rightarrow B \quad v(L) = v(B), s(B) = s(L) \\
L_1 & \rightarrow L_2 B \quad v(L_1) = v(L_2) + v(B), s(B) = s(L_1) \\
& \quad s(L_2) = s(L_1) + 1 \\
N & \rightarrow L \quad v(N) = v(L), s(L) = 0
\end{align*}
\]