Introduction to Control Flow Analysis and Slicing

Course Notes for CISC422/853 (Winter 2009)

Juergen Dingel (dingel@cs.queensu.ca)

March 3, 2009

Contents

1 Control flow analysis

The purpose of control flow analysis typically is to determine which statements may be executed after some other statement. Control flow analysis is used for compiler optimization and data-flow analysis. Control flow analysis typically computes a *control flow graph (CFG)*. A node in a CFG represents an atomic statement or a boolean condition. An edge in a CFG represents the possible flow of control.

1.1 For imperative languages

For imperative languages without first-order procedures (procedures cannot be passed as arguments or returned as results), control flow analysis usually is very straight-forward. Iteration statements like **while** and **repeat** statements cause branching and looping in the graph, whereas conditional statements only cause branching. The program

$$
\begin{array}{lll}
C_1 \equiv & 1: & a{:=}5; \\
& 2: & c{:=}1; \\
& 3: & \textbf{while } c \leq a \textbf{ do} \\
& 4: & \quad c{:=}c + a \\
& 5: & \textbf{end;} \\
& 6: & a{:=}a - c; \\
& 7: & c{:=}0
\end{array}
$$

for instance, has the control flow graph given in Figure 1. The line numbers in a program are called *locations*. We will assume that every location l in a program P corresponds uniquely to a node in the control flow graph of P. For convenience, we will use locations to refer to nodes and vice versa.

Note that some control flow analyses group sequences of nodes that are connected linearly and without branching into a single node. For instance, the sequence of nodes in Figure 1 with locations 1, 2, and 3 would be replaced by a single node. We will not consider this kind of grouping here.

The algorithm for computing the control flow graph from the abstract syntax tree (AST) of a program is given in Figure 2. Each node in the CFG has a list of predecessor (*preds*) and a list of successors (*succs*). The key idea of the algorithm is to represent a CFG by a pair of nodes *(first, last)*. Node *first* forms the entry into the CFG while node *last* is the exit of the CFG. The method *toCFG(Stmt s)* is defined inductively over the structure of s. If s is an assignment, a single node is returned. If s is a sequential composition $s_1\,;s_2$, the CFGs for s_1 and s_2 are computed recursively. The CFG for $s_1;s_2$ is then obtained by connecting the exit of s_1 with the entry of s_2, that is, the entry of s_2 is added to the successors of the exit of s_1 and the exit of s_1 is added to the predecessor of the entry of s_2. The remaining cases are similar.

The control flow graph G of a program C has two important properties:

- Every program execution of C will have a corresponding path in G.

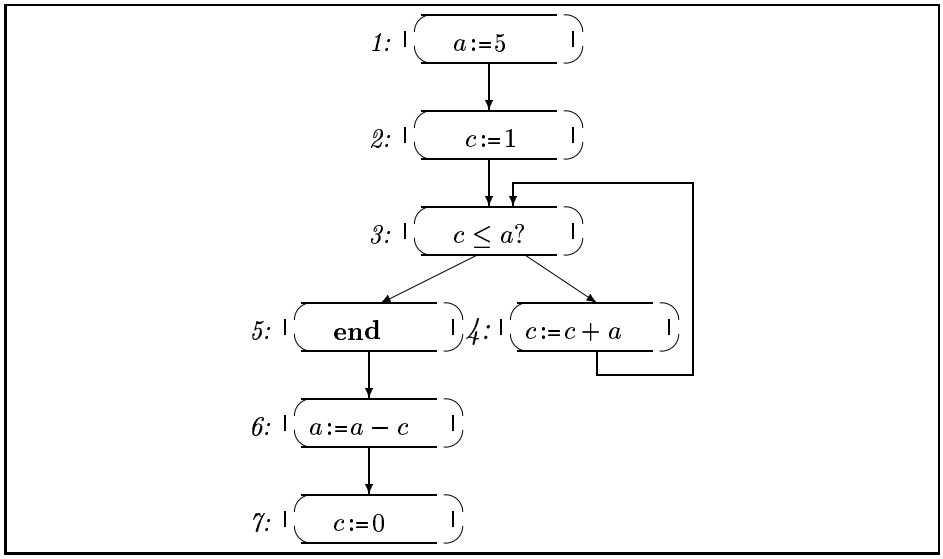

Figure 1: Control flow graph of program C_1

- However, a path in G does not necessarily correspond to a program execution of C.

Note that branching conditions in CFGs remain uninterpreted. A path might thus represent an unfeasible execution. The program

```
1:  x:=1;
2:  if x = 1 then
3:     x:=2
4:  end
```

for instance, only has one execution, but its control flow graph contains two paths.

1.2 For functional and object-oriented languages

For higher-order functional languages and object-oriented languages control flow analysis is a lot harder. To see this, consider the following functional and object-oriented program fragments.

```
public class Node {
  public Vector preds, succs;
  public link (Node n1, Node n2) {
    n1.succs.add(n2) ; n2.preds.add(n1)
  }
  ...
}
public class Cfg {
  public Node first, last;
  public Cfg(Node f, Node l) {
    first = f; last = l;
} }
public Cfg toCfg(Stmt s){
  if (s is an assignment x:=e) then
    Node n = new Node(x, e);
    return new Cfg(n, n);
  else if (s is sequential composition s1 ; s2) then
    Cfg cfg1 = toCfg(s1);
    Cfg cfg2 = toCfg(s2);
    link(cfg1.last, cfg2.first);
    return new Cfg(cfg1.first, cfg2.last);
  else if (s is if b then s1 else s2) then
    Cfg cfg1 = toCfg(s1);
    Cfg cfg2 = toCfg(s2);
    Node testNode = new Node(b);
    link(testNode, cfg1.first);
    link(testNode, cfg2.first);
    Node endNode = new Node("END");
    link(cfg1.last, endNode);
    link(cfg2.last, endNode);
    return new Cfg(testNode, endNode)
  else if (s is while b do s1) then
    Cfg cfg = toCfg(s1);
    Node firstNode = cfg.first;
    Node testNode = newNode(b);
    link(testNode, cfg.first);
    link(cfg.last, testNode);
    Node endNode = new Node("END");
    link(testNode, endNode);
    return new Cfg(testNode, endNode);
  }
```

Figure 2: Algorithm to compute CFG from AST

```
let
  app  = fn f, x in (f x)
  add1 = fn y in (y + 1)
  add2 = fn y in (y + 2)
in
  (app add1 0) + (app add2 0)
end
```

```
class A {
  public void m() {...}
}
class subA {
  public void m() extends A {...}
}
class subsubA {
  public void m() extends subA {...}
}
...a.m()...
```

In the functional program on the left, the function that gets applied to x in the body of function *app* depends on the context in which *app* is called. In the object-oriented program on the right, the version of m that is executed (if any) depends on the type of a. The presence of higher-order functions and object-oriented polymorphism give the control flow more variability and make it much harder to perform control flow analysis.

2 Program slicing

A program slice consists of the parts of a program that (potentially) affect the values computed at some point of interest, referred to as a slicing criterion. The task of computing program slices is called program slicing. The original definition of a program slice was presented by Weiser in 1979 [Wei79]. Since then, various slightly different notions of program slices have been proposed, as well as a number of methods to compute them. An important distinction is that between a *static* and a *dynamic* slice. The former notion is computed without making assumptions regarding a program's input, whereas the latter relies on some specific test case. Procedures, unstructured control flow (eg, GOTO), composite datatypes and pointers, and interprocess communication each require a specific solution. In this introduction we will only consider the simplest case: static slicing of structured imperative programs without nonscalar variables (eg, arrays or pointers), procedures, or interprocess communication.

2.1 Basic definitions for static slicing

We have seen how (mutually recursive) dataflow equations can be used to describe conveniently various static program analyses like reaching definitions, available expressions, or live variables. In each case, the result of the analysis was obtained from a solution of these equations. The desired solution to the equations was given by the least (in case of reaching definitions and liveness) or the greatest (in case of available expressions) fixed point of the (mutually recursive) functions described by the equations and was obtained through straightforward fixed point iteration that we have already seen in the context of model checking.

Since static program slicing is nothing else but another static program analysis technique, Weiser's original definition in [Wei79] also is based on the iterative solution of a set of dataflow equations. However, while the static program analyses we have already seen are concerned with determining certain information about each node in the Control Flow Graph (CFG) of the program, the goal of program slicing is to identify parts of the program that are (ir)relevant to the values of certain variables at certain program points. Another difference is that the equations used to describe a program slice do not involve *gen* and *kill* functions. Intuitively, this is because the information generated and killed at each node does depend not only on the statement executed at the node, but also on the slicing criterion and the surrounding program context.

Definition 2.1 (Basic definitions)
Let P be a program.

1. **Slice.** A slice S of P is an executable program that is obtained from P by deleting zero or more statements.
2. **Slicing criterion.** A slicing criterion consists of a pair (n, V) where n is a node in the CFG of P and V is a subset of the variables in P.
3. **Slice with respect to criterion.** Let (n, V) be a slicing criterion of P. A slice S of P is called a slice of P with respect to criterion (n, V), if it contains the statement at node n and whenever P halts for a given input,
   - S also halts for that input, and
   - S computes the same values for the variables in V whenever the statement corresponding to the node n is executed.
4. **Minimal slice.** A slice is called minimal, if no other slice for the same criterion contains fewer statements. □

Theorem 1 Minimal slices are not necessarily unique and the problem of determining whether a given slice is minimal is undecidable.

Proof: Omitted. ■

Just like for the other program analyses we've discussed, the fact that the underlying problem is undecidable destroys all hope for an algorithm that always terminates with the exact solution, that is, with a minimal slice. We therefore have to settle for a (hopefully still informative) *approximation* to the exact solution.

2.2 Relevant variables and statements

The most important notion for computing slices is that of a relevant variable. All other necessary notions build on that. This section presents the definition of relevant variables and shows how it can be used to compute slices. Section 2.3 will then show how the relevant variables can be computed.

Definition 2.2 (Relevant variables)
Let i be a node in the CFG of a program P and let $C = (n, V)$ be a criterion. We say that a variable v is *relevant* at a node i with respect to C if the value of v right before execution of the statement in i may influence the value of at least one variable in V at node n. □

In other words, given an execution of the program that is about to go through node i, changing the value of v right before i is entered, may also change the value of at least one variable in V at n. Consider, for instance, the following program P_1

```
1 :  z:=w;
2 :  y:=z;
3 :  x:=y;
4 :  output(x)
```

with the criterion $C_1 = (4, \{x\})$. The following table summarizes which variables are relevant at which node in the CFG of P_1.

node i	4	3	2	1
variables relevant at i	$\{x\}$	$\{y\}$	$\{z\}$	$\{w\}$.

Clearly, changing the value of y right before node 3 will influence the value of x at 4, that is, y is relevant at 3 with respect to C_1. However, changing the value of x right before node 1, will never have any effect on the value of x at 4, that is, x is not relevant at 1 with respect to C_1. As another example, consider the program P_2

```
1 :  if w = tt then
2 :    z:=tt
       else
3 :    d:=1;
4 :  if z = tt then
5 :    y:=tt
       else
6 :    d:=d+1;
7 :  if y = tt then
8 :    x:=tt
       else
9 :    d:=d+1;
10 :  output(x)
```

with the criterion $C_2 = (10, \{x\})$. The relevant variables are given by the following table.

node i	10	9	8	7	6	5	4	3	2	1
rel.vars.	$\{x\}$	$\{x\}$	$\{\}$	$\{x,y\}$	$\{x,y\}$	$\{x\}$	$\{x,y,z\}$	$\{x,y,z\}$	$\{x,y\}$	$\{x,y,z,w\}$

Once we have the relevant variables, computing the corresponding slice is easy.

Definition 2.3 (Relevant statements)
Given a program P with node i and a criterion C, we say that

- an assignment at node i in P is *relevant* with respect to C if it defines a variable that is relevant at at least one successor of i with respect to C,
- an **if** b **then** P_1 **else** P_2 statement at node i in P is *relevant* with respect to C if either P_1 or P_2 contain at least one relevant statement,
- a **while** b **do** P' statement at node i in P is *relevant* with respect to C if P' contains at least one relevant statement. □

For instance, the following statements of P_1 are relevant with respect to C_1.

```
1 :  z:=w;
2 :  y:=z;
3 :  x:=y;
```

Similarly, the statements

```
1 :  if w = tt then
2 :    z:=tt
4 :  if z = tt then
5 :    y:=tt
7 :  if y = tt then
8 :    x:=tt
```

from P_2 are relevant with respect to C_2.

Computing slices from relevant variables and statements

It turns out that if we take the statement mentioned in the criterion and all relevant statements, then we get a slice.

Theorem 2 Given a program P and a criterion $C = (n, V)$, the statement at n together with the set of statements of P relevant with respect to C is a slice of P with respect to C.

Proof: Need to show that requirements in Definition 2.1 are met. Omitted. ■

The algorithm to be presented in Section 2.4 will compute the slice of a program P with respect to criterion $C = (n, V)$ as follows:

1. First, the variables of P relevant with respect to C are computed.
2. Then, the relevant variables are used to determine the set of statements of P relevant with respect to C.
3. Finally, the slice is obtained by adding the statement in node n to the set of relevant statements.

For each of the example programs and criteria given above, convince yourself that above three steps do indeed compute slices.

While the implementation of the second step is straight-forward, the implementation of the first step is not. The next section describes the sets of relevant variables and statements using dataflow equations. The implementation of the first step can then be read off these equations in a way similar to the one already used for the program other analyses.

2.3 Dataflow equations for relevant variables and statements

We need to identify a subset of the relevant variables and statements called directly relevant variables and statements.

Directly relevant variables and statements

Intuitively, the sets of directly relevant variables and statements only capture which variables and statements directly influence the criterion without taking the control flow into account.

In the following, let i, j range over nodes in the CFG and let the notation $i \longrightarrow_{CFG} j$ express that there is an edge from node i to node j in the CFG, that is, j is a successor of i. Let $Def(i)$ denote the set of variables defined (i.e., assigned to) at i. Note that in the presence of aliases, a single assignment can define multiple variales. Also, let $Use(i)$ denote the set of variables used (i.e., read) at i.

2.3.1 Directly relevant variables $R^0_C(i)$

Given a slicing criterion $C = (n, V)$ and a node i, the set of directly relevant variables at i, $R^0_C(i)$, is defined as[1]:

- all criterion variables are directly relevant at the criterion node, that is, $R^0_C(i) = V$ if $i = n$.
- otherwise, v is directly relevant at i, if i has a successor j such that
  - v is either directly relevant at j and not defined (assigned to) in i, or
  - v is used (evaluated) in i and i defines a variable that is directly relevant at j.

  Formally, for all nodes i in the CFG, $v \in R^0_C(i)$ if there exists a node j such that $i \longrightarrow_{CFG} j$ and
  - $v \in R^0_C(j)$ and $v \notin Def(i)$, or
  - $v \in Use(i)$ and $Def(i) \cap R^0_C(j) \neq \{\}$.

[1] The superscript in R^0_C indicates that no branching statements are considered.

Or, even more formally,

$$R^0_C(i) \quad = \quad \begin{cases} V, & \textit{if } i{=}n \\ \{v \mid \exists j.i \longrightarrow_{CFG} j \wedge \big((v \in R^0_C(j) \wedge v \notin \textit{Def}(i)) & \\ \quad \vee (v \in \textit{Use}(i) \wedge \textit{Def}(i) \cap R^0_C(j) \neq \{\})\big)\}, & o/w \end{cases} \tag{1}$$

for all i. Note that if node i defines exactly one variable, then the condition $\textit{Def}(i) \cap R^0_C(j) \neq \{\}$ can be simplified to $\textit{Def}(i) \in R^0_C(j)$.

Intuitively, if x is directly relevant at node i, then the value of x right before execution of i may influence the values of the variables in V at n without taking the control flow of the program into account. For instance, let P_3 be the program

```
1 :  w:=0;
2 :  if x = r then
3 :    y:=y + 1;
     else
4 :    z:=0;
5 :  x:=z + w;
6 :  output(x)
```

where the scope of the conditional is indicated by indentation and let C_3 be the slicing criterion $(6, \{x, y\})$. We have

node i	6	5	4	3	2	1
$R^0_C(i)$	$\{x, y\}$	$\{w, y, z\}$	$\{w, y\}$	$\{w, y, z\}$	$\{w, y, z\}$	$\{y, z\}$.

Variable w is not directly relevant at node 1, because the value of w right before 1 does not influence the value of x or y at 6. Variable y is directly relevant at node 3, because the value of y right before execution of 3 influences the value of y at node 6 even without taking boolean conditions into account. The fact that the variables y and z are relevant at the first node indicates that these variables are uninitialized and that the value of x or y at 6 depends on them. Note that variables r and x are not directly relevant at any of the nodes. This is because, as mentioned above, direct relevance ignores the control flow. But, clearly, the values of r and x before nodes 1 and 2 may influence the value of y at node 6. Therefore, r and x are relevant at 1 and 2 in the sense of Definition 2.2. In other words, the notion of direct relevance does not quite capture the notion of relevance of Definition 2.2.

2.3.2 Directly relevant statements S^0_C

A statement at node i is directly relevant if it defines a variable that is directly relevant at a successor j of i.

$$S^0_C \quad = \quad \{i \mid \exists j.i \longrightarrow_{CFG} j \text{ and } \textit{Def}(i) \cap R^0_C(j) \neq \{\}\}$$

For instance, program P_3 contains the following directly relevant statements with respect to criterion $C_3 = (6, \{x, y\})$

```
1 :  w:=0;
3 :    y:=y + 1;
4 :    z:=0;
5 :  x:=z + w;
```

Note that the statement at node 6 together with the above statements does not form a slice of P_3 with respect to C_3. This is because the value of y at 6 in the resulting program may be different from the value of y at 6 in P_3. Thus, the second condition of Definition 2.1.3 is violated.

Relevant variables and statements

As mentioned above, the computation of the directly relevant variables ignores the variables that influence the variables in the criterion by influencing boolean conditions in **if** or **while** statements. For instance, variables r and x in program P_3 above are not directly relevant at nodes 2 or 1. But, as already argued above, both r and x are relevant at nodes 1 and 2 in the sense of Definition 2.2. The following definition of relevant branch statements will help to remedy this discrepancy.

2.3.3 Relevant branch statements B_C

We call the boolean tests b in **if** and **while** statements *branch statements.* The scope of a branch statement is either the **then** and the **else** branch or the loop body. A branch statement is relevant, if its scope contains a relevant statement.

$$B_C \quad = \quad \{b \mid b \textit{ is branch statement and there exists a node } i \in S_C^{>0}, \textit{ such that } i \textit{ is in the scope of } b\}$$

For instance, branch statement 2 in program P_3 is relevant, because it has two relevant statements in its scope.

2.3.4 Relevant variables $R_C^{>0}(i)$

A variable is relevant at i if it is directly relevant at i or it is directly relevant for one of the relevant branch statements.

$$R_C^{>0}(i) \quad = \quad R_C^0(i) \cup \bigcup_{b \in B_C} R^0_{(b, Use(b))}(i)$$

for all i in the CFG[2]. For instance, for P_3 and C_3 we have

node i	6	5	4	3	2	1
$R_C^{>0}(i)$	$\{x, y\}$	$\{w, y, z\}$	$\{w, y\}$	$\{w, y, z\}$	$\{r, w, x, y, z\}$	$\{r, x, y, z\}$.

[2]The superscript in $R_C^{>0}$ indicates that branching statements are considered.

The above table is obtained from the previous one by including r and x into the relevant variables at nodes 1 and 2.

2.3.5 Relevant statements $S_C^{>0}$

A statement is relevant at i if it is a relevant branching statement or if it defines a variable that is relevant a successor of i.

$$S_C^{>0} \quad = \quad B_C \cup \{i \mid i \longrightarrow_{CFG} j \text{ and } \mathit{Def}(i) \cap R_C^{>0}(j) \neq \{\}\}$$

For instance, all statements of program P_3 are relevant with respect to C_3.

The dataflow equations

The equations for directly relevant and relevant variables and statements together form the dataflow equations that describe the slicing problem. As before, let C be the criterion (n, V).

For all nodes i in the CFG, we have

$$R_C^0(i) = \begin{cases} V, & \textit{if } i{=}n \\ \{v \mid \exists j.i \longrightarrow_{CFG} j \wedge \big((v \in R_C^0(j) \wedge v \notin \mathit{Def}(i)) \\ \qquad \vee (v \in \mathit{Use}(i) \wedge \mathit{Def}(i) \cap R_C^0(j) \neq \{\})\big)\}, & o/w \end{cases} \quad (2)$$

$$S_C^0 = \{i \mid \exists j.i \longrightarrow_{CFG} j \wedge \mathit{Def}(i) \cap R_C^0(j) \neq \{\}\} \quad (3)$$

$$B_C = \{b \mid b \textit{ is branch statement and there exists } i \in S_C^{>0}, \textit{ such that } i \textit{ is in the scope of } b\} \quad (4)$$

$$R_C^{>0}(i) = R_C^0(i) \cup \bigcup_{b \in B_C} R^0_{(b,\, \mathit{Use}(b))}(i) \quad (5)$$

$$S_C^{>0} = B_C \cup \{i \mid \exists j.i \longrightarrow_{CFG} j \wedge \mathit{Def}(i) \cap R_C^{>0}(j) \neq \{\}\} \quad (6)$$

A solution of the above system of equations consists of two sets of variables for each node i (the directly relevant and the relevant variables at i), two sets of statements (the directly relevant and relevant statements), and a set of branching statements, such that the substitution of the variables in the equations (S_C^0, R_C^0, B_C, $R_C > 0$, and $S_C^{>0}$) by their corresponding sets makes the all equations true.

Besides the definitions in Section 2.2, the system of equations form a second description of relevant variables and statements. We now have to argue that the two descriptions coincide.

Theorem 3 Given a program P and a criterion C, a set of variables is a solution to equation (5) if and only if it is relevant with respect to C in the sense of Definition 2.2.

Proof: Omitted. ■

Corollary 2.1 Given a program P and criterion $C = (n, V)$,

1. a set of statements of P is a solution to equation (5) if and only if it is relevant with respect to C in the sense of Definition 2.3.

2. solutions of equation (6) together with the statement in node n are slices of P. The smallest such solution is the minimal computed slice. □

Based on the above equations, we will now present an algorithm to compute minimal slices.

2.4 Solving the dataflow equations

We first compute the directly relevant variables and statements.

Computing the directly relevant variables and statements

1. For each node i in the CFG, initialize $R^0_C(i)$ with $\{\}$.

2. For each node i in CFG, compute the directly relevant variables at i, $R^0_C(i)$, using equation (2).

3. If there exists a node j such that the previous step has changed $R^0_C(j)$, then go back to 2. Else, stop. $R^0_C(i)$ contains the directly relevant variables at i for all i.

4. Compute the directly relevant statements S^0_C using equation (3).

Note that Step 2 is not specific about the order in which the nodes are being considered. It turns out that this order influences the number of iterations necessary to reach the fixed point. If the sets $R^0_C(i)$ are computed in topologically sorted order, then each $R^0_C(i)$ only needs to be considered once and one iteration suffices. See, for instance, program P_3 above. The reason is that the topological order implies that the system of equations (1) for R^0_C is not mutually recursive. Therefore it can be solved using standard algebraic substitution. Intuitively, the topological order ensures that the information necessary for the updates for node i has already been computed when i is being considered. However, the nodes of a graph can only be topologically sorted if and only if the graph is acyclic. If the graph is cyclic, then each $R^0_C(i)$ may have to be considered several times before the correct value is found. In other words, we have to do fixed point iteration. The reason is that if the CFG is cyclic, then the system of equations for R^0_C is mutually recursive which means that it cannot be solved using straight-forward algebraic substitution. Note that we have already encountered this situation in the previous program analyses.

Computing the relevant variables and statements

Since the equations for relevant variables and branch statements are mutually recursive, we again need to resort to fixed point iteration. Starting with the

directly relevant variables and statements as our initial approximation, the relevant variables and statements are computed iteratively. In each iteration, the current approximation is used to obtain a possibly better (but never worse) one. Once all relevant variables have been determined, no further improvement is possible, the iteration stops and the corresponding slice is output.

1. Compute the directly relevant variables and statements using the algorithm in Section 2.4.
2. Initialize $R_C^{\geq 0}(i)$ for all i and $S_C^{\geq 0}$ with $\{\}$.
3. For each node i in CFG, compute the relevant variables at i, $R_C^{\geq 0}(i)$, using equation (5).
4. Compute the relevant statements $S_C^{\geq 0}$ using equation (6).
5. Compute the relevant branch statements B_C using equation (4).
6. If there exists a node j such that $R_C^{\geq 0}(j)$ has changed during Step 3, then goto back to 3. Else, stop. $S_C^{\geq 0} \cup \{n\}$ contains the (approximation of the) minimal slice with respect to $C = (n, V)$.

Theorem 4 The above algorithm always terminates with the smallest solution of the equations given in Section 2.3.

Proof: None of the sets updated in each iteration ever get smaller. Since every program has a finite number of variables and statements, both of the above iterations must eventually terminate. The proof that the least solution is computed is omitted. ■

According to Weiser, the above algorithm is $O(n_v \times n_i \times n_e)$ where n_v is the number of variables in the program, n_i is the number of nodes in the CFG, and and n_e is the number of edges in the CFG [Wei79]. Note that the complexity is independent of the number of variables in the slicing criterion.

2.5 Examples

2.5.1 Example 1

As a warm-up, reconsider program P_1

```
1 :  z:=w;
2 :  y:=z;
3 :  x:=y;
4 :  output(x)
```

with the criterion $C_1 = (4, \{x\})$. Intuitively, the value that x ends up having at node 4 is same as that of variable w at node 1, because it "flows" through the variables z and y to x. None of the statements can be removed without

creating the possibility of changing the value of x at 4. The program is its own minimal slice with respect to criterion C_1. This is the result our algorithm should compute. Let's see if it really does. First, we have to compute the directly relevant variables $R_C^0(i)$. Suppose we consider the nodes against the control flow (from largest node id to smallest node id). Then, we get

node i	4	3	2	1
$R_C^0(i)$	$\{x\}$	$\{y\}$	$\{z\}$	$\{w\}$.

The set of directly relevant statements S_C^0, that is, the set of nodes that define a variable that is directly relevant at a successor, is

$$S_C^0 \quad = \quad \{1, 2, 3\}.$$

Since the program does not contain any branch statements, the indirectly relevant variables are identical to the directly relevant ones, that is, $R_C^{\geq 0}(i) = R_C^0(i)$ for all i, which means that R_C^0 and S_C^0 are fixed points and $S_C^0 \cup \{4\}$ constitutes the minimal slice with respect to C_1. The algorithm thus validates our result from Section 2.2. Note that had we considered the nodes in opposite direction, that is, in direction of the control flow, three iterations would have been necessary to reach the fixed point.

2.5.2 Example 2

Let us now consider the program P_4

```
1 :  v:=3;
2 :  w:=42;
3 :  x:=5 + v;
4 :  z:=w;
5 :  y:=z;
6 :  output(x)
```

with the criterion $C_4 = (6, \{x\})$. We get

node i	6	5	4	3	2	1
$R_C^0(i)$	$\{x\}$	$\{x\}$	$\{x\}$	$\{v\}$	$\{v\}$	$\{\}$

The fact that the first node in the CFG does not have any relevant variables indicates that the value of x at 6 does not depend on any uninitialized variables. Only the statements in nodes 1 and 3 define a variable that is directly relevant to one of their successors. Thus,

$$S_C^0 \quad = \quad \{1, 3\}.$$

Since the program does not contain any branch statements, the set of indirectly relevant variables is the same as the set of directly relevant variables, that is,

$R^0_C = R^{>0}_C$ which implies $S^0_C = S^{>0}_C$ and means that the program

```
1 :  v:=3;
2 :
3 :  x:=5 + v;
4 :
5 :
6 :  output(x)
```

is the computed minimal slice.

2.5.3 Example 3

In the next example, we reconsider program P_2

```
1 :  if w = tt then
2 :    z:=tt
     else
3 :    d:=1;
4 :  if z = tt then
5 :    y:=tt
     else
6 :    d:=d + 1;
7 :  if y = tt then
8 :    x:=tt
     else
9 :    d:=d + 1;
10 :  output(x)
```

with the criterion $C_2 = (10, \{x\})$. We get

node i	10	9	8	7	6	5	4	3	2	1
$R^0_C(i)$	$\{x\}$	$\{x\}$	$\{\}$	$\{x\}$	$\{x\}$	$\{x\}$	$\{x\}$	$\{x\}$	$\{x\}$	$\{x\}$

and

$$S^0_C \quad = \quad \{8\}.$$

Note that 8 is the only node at which x is not directly relevant. This is because 8 defines x and does not use x. Intuitively, the value of x before execution of 8 is irrelevant for the value of x at node 10. The fact that 8 is in S^0_C means that the algorithm has determined that statement 8 is relevant for the value of x at 10 and thus needs to be included in the slice.

Since none of the examples in this section contained any branching statements, the fixed point was always reached after one iteration. In the case of Program P_2 more iterations will be necessary.

- Iteration 1: At the beginning of the first iteration, we compute the relevant branch statements B_C. Node 8 is under the scope of branch statement 7.

$$B_C \quad = \quad \{7\}.$$

When computing the indirect relevant variables $R_C^{>0}$, we include the variables that influence branch condition 7, because it in turn influences the value of x at 10. Thus,

node i	10	9	8	7	6	5	4	3	2	1
$R_C^{>0}(i)$	$\{x\}$	$\{x\}$	$\{\}$	$\{x,y\}$	$\{x,y\}$	$\{x\}$	$\{x,y\}$	$\{x,y\}$	$\{x,y\}$	$\{x,y\}$

and

$$S_C^{>0} \quad = \quad \{5,7,8\}.$$

Since $S_C^0 \neq S_C^{>0}$, we continue with the next iteration.

- Iteration 2: When recomputing B_C, the branching statements at node 4 now becomes relevant because the relevant statement 5 is in its scope.

$$B_C \quad = \quad \{4,7\}.$$

Thus, also the variables that influence the condition in 4 must be considered relevant.

node i	10	9	8	7	6	5	4	3	2	1
$R_C^{>0}(i)$	$\{x\}$	$\{x\}$	$\{\}$	$\{x,y\}$	$\{x,y\}$	$\{x\}$	$\{x,y,z\}$	$\{x,y,z\}$	$\{x,y\}$	$\{x,y,z\}$

and

$$S_C^{>0} \quad = \quad \{2,4,5,7,8\}.$$

Since the set $S_C^{>0}$ has changed in this iteration, we need to continue.

- Iteration 3: Again, the increase in relevant variables means an increase in relevant branch statements.

$$B_C \quad = \quad \{1,4,7\}.$$

The indirectly relevant variables and statements change again.

node i	10	9	8	7	6	5	4	3	2	1
$R_C^{>0}(i)$	$\{x\}$	$\{x\}$	$\{\}$	$\{x,y\}$	$\{x,y\}$	$\{x\}$	$\{x,y,z\}$	$\{x,y,z\}$	$\{x,y\}$	$\{x,y,z,w\}$

and

$$S_C^{>0} \quad = \quad \{1,2,4,5,7,8\}.$$

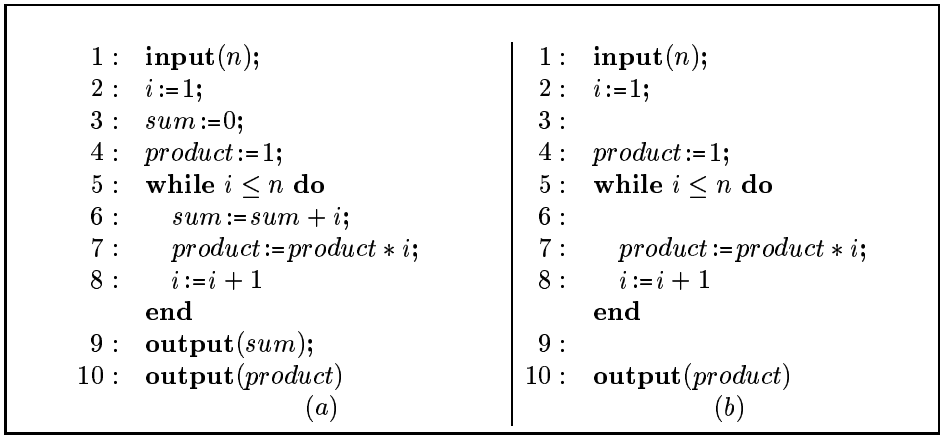

Figure 3: (a) Example program. (b) A slice of the program with respect to the criterion $(10, product)$.

Finally, a fixed point is reached and we can terminate. Program

```
 1 : if w = tt then
 2 :   z:=tt
 4 : if z = tt then
 5 :   y:=tt
 7 : if y = tt then
 8 :   x:=tt
10 : output(x)
```

is the computed minimal slice. Our algorithm has correctly determined that the **else** branches of the conditionals are irrelevant for the value of x at node 10 and our result from Section 2.2 is validated.

2.5.4 Example 4

As a final example, consider the left program in Figure 3. The sets of relevant variables arising during the computation are given in Figure 4. The fixed point is reached after two iterations. The right program in Figure 3 is the computed slice.

References

[Wei79] M. Weiser. *Program slices: Formal, psychological, and practical investigations of an automatic program abstraction method.* PhD thesis, University of Michigan, 1979.

node i	$R_C^0(i)$	$R_C^{>0}(i)$
1	$\{\}$	$\{\}$
2	$\{\}$	$\{n\}$
3	$\{i\}$	$\{i, n\}$
4	$\{i\}$	$\{i, n\}$
5	$\{product, i\}$	$\{product, i, n\}$
6	$\{product, i\}$	$\{product, i, n\}$
7	$\{product, i\}$	$\{product, i, n\}$
8	$\{product, i\}$	$\{product, i, n\}$
9	$\{product\}$	$\{product\}$
10	$\{product\}$	$\{product\}$

$$S_C^0 = \{2, 4, 7, 8\}$$
$$B_C = \{5\}$$
$$S_C^{>0} = \{1, 2, 4, 5, 7, 8\}$$

Figure 4: Final results of the application of the slicing algorithm to the left program in Figure 3 and slicing criterion $(10, \{product\})$.