Program Slicing
The final project survey

• Quick logistics note

  • If you haven’t filled the final projects group survey, and especially if you want me to pair you with a team, please fill the survey soon!

  • (This is for getting a snapshot of where we are now, not for immediately before the team formation deadline!)

• Find link in the slack!
Reading reflection

• What did the two tools you saw in the demo videos have in common?

• When you figure out that a program you’re debugging is producing a wrong output, what’s your next step?
The key common feature: program slicing!

Examples of Slices

The original program:

```plaintext
1 BEGIN
2 READ(X,Y)
3 TOTAL := 0.0
4 SUM := 0.0
5 IF X = 1
6 THEN SUM := Y
7 ELSE BEGIN
8 READ(Z)
9 TOTAL := X*Y
10 END
11 WRITE(TOTAL,SUM)
12 END.
```

Slice on the value of Z at statement 12.

```plaintext
BEGIN
READ(X,Y)
IF X < 1
THEN ELSE READ(Z)
END.
```

Slice on the value of X at statement 9.

```plaintext
BEGIN
READ(X,Y)
END.
```

Slice on the value of TOTAL at statement 12.

```plaintext
BEGIN
READ(X,Y)
TOTAL := 0
IF X <= 1
THEN ELSE TOTAL := X*Y
END.
```
The same example, but maybe a bit more familiar looking!
let's diy!
Ok, let’s get a look at this AST thing

Our Python program (the one we’re analyzing, not the one we’re running)

```python
sum_acc = 0
prod_acc = 1
i = 1
while (i < 11):
    sum_acc = sum_acc + i
    prod_acc = prod_acc * i
    i = i + 1
print("sum", sum_acc)
print("prod", prod_acc)
```

Here’s the one we’re running...

```python
code = open(filename).read()
tree = ast.parse(code)
astpretty.pprint(tree)
```
Next, we need to know how control flows through the program

Enter…the control flow graph (CFG)

We’ll build up a graph representing all the paths we could take through the program during execution

Another entry in our theme of ‘there are so many ways to represent programs’!
CFGs

Program to analyze

```
1  sum_acc = 0
2  prod_acc = 1
3  i = 1
4  while (i < 11):
    5     sum_acc = sum_acc + i
    6     prod_acc = prod_acc * i
    7     i = i + 1
8  print("sum", sum_acc)
9  print("prod", prod_acc)
```

The CFG!
But how do we get the slice from this thing?

Starting from a CFG, we’ll compute data flow information about the set of relevant variables at each node

<table>
<thead>
<tr>
<th>n</th>
<th>Statement</th>
<th>ref(n)</th>
<th>def(n)</th>
<th>relevant(n)</th>
</tr>
</thead>
</table>

We’ll use this ^ and this ^ to figure out this ^

Referenced at node n

Defined at node n
Program Slicing: Straight-Line Code

Slice for node \( n \) and variables \( V \)

1. Initialize the relevant sets of all nodes to the empty set.
2. Insert all variables of \( V \) into \( \text{relevant}(n) \).
3. For \( n \)'s immediate predecessor \( m \), compute \( \text{relevant}(m) \) by:
   
   // first exclude all variables defined at \( m \) (because we’re overwriting it)
   \[
   \text{relevant}(m) := \text{relevant}(n) - \text{def}(m)
   \]
   // if \( m \) defines a variable that’s relevant at \( n \)
   if \( \text{def}(m) \) in \( \text{relevant}(n) \) then
     // include the variables that are referenced at \( m \)
     \[
     \text{relevant}(m) := \text{relevant}(m) \cup \text{ref}(m)
     \]
     include \( m \) in the slice
   end

4. Repeat (3) for \( m \)'s immediate predecessors, and work backwards in the CFG until we reach the start node or the relevant set is empty
<table>
<thead>
<tr>
<th>n</th>
<th>Statement</th>
<th>ref(n)</th>
<th>def(n)</th>
<th>relevant(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>b = 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>c = 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>d = 3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>a = d</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>d = b + d</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>b = b + 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>a = b + c</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>print a</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Bolded n are included in the slice**

**slice for <8, {a}>**

Step 2: \( \text{relevant}(8) = \{a\} \)

Step 3: \( \text{relevant}(7) = \text{relevant}(8) - \text{def}(7) = \{a\} - \{a\} = \{\} \)

Step 3: \( \text{relevant}(7) = \text{relevant}(7) \cup \text{ref}(7) = \{\} \cup \{b, c\} = \{b, c\} \)

Since node 7 defines a variable relevant at node 8, it is included into the slice.

Step 3: \( \text{relevant}(6) = \text{relevant}(7) - \text{def}(6) = \{b, c\} - \{b\} = \{c\} \)

Step 3: \( \text{relevant}(6) = \text{relevant}(6) \cup \text{ref}(6) = \{c\} \cup \{b\} = \{b, c\} \)

Since node 6 defines a variable relevant at node 7, it is included into the slice.

Step 3: \( \text{relevant}(5) = \text{relevant}(6) - \text{def}(5) = \{b, c\} - \{d\} = \{b, c\} \)

Step 3: \( \text{relevant}(4) = \text{relevant}(5) - \text{def}(4) = \{b, c\} - \{a\} = \{b, c\} \)

Step 3: \( \text{relevant}(3) = \text{relevant}(4) - \text{def}(3) = \{b, c\} - \{d\} = \{b, c\} \)

Step 3: \( \text{relevant}(2) = \text{relevant}(3) - \text{def}(2) = \{b, c\} - \{c\} = \{b\} \)

Step 3: \( \text{relevant}(2) = \text{relevant}(2) \cup \text{ref}(2) = \{b\} \cup \{\} = \{b\} \)

Since node 2 defines a variable relevant at node 3, it is included into the slice.

Step 3: \( \text{relevant}(1) = \text{relevant}(2) - \text{def}(1) = \{b\} - \{b\} = \{\} \)

Step 3: \( \text{relevant}(1) = \text{relevant}(1) \cup \text{ref}(1) = \{\} \cup \{\} = \{\} \)

Since node 1 defines a variable relevant at node 2, it is included into the slice.
What will happen if we add an if statement into our program?

- Any guesses?
Moving towards handling control flow...

• We have to extend our earlier approach to:
  • If we add a node \( m \) to our slice:
    • also add the control set of \( m \) to our slice
    • (the control set is the set of predicates that directly control its execution)
    • for each node \( c \) included based on being in the control set:
      • make a new slice! Starting at node \( c \) for variables \( \text{ref}(c) \). The original slice (for \(<n, V>\)) will now also include all nodes in the slice for \(<c, \text{ref}(c)>\)
  • Union the relevant sets (e.g., \( \text{relevant}(m_1) \) and \( \text{relevant}(m_2) \)) for cases where we have multiple descendants with a shared predecessor
    • (Remember that once we have control flow, we can have multiple descendants!)
slice for <11, \{a\}>

<table>
<thead>
<tr>
<th>n</th>
<th>Statement</th>
<th>ref(n)</th>
<th>def(n)</th>
<th>control(n)</th>
<th>relevant(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>b = 1</td>
<td></td>
<td>b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>c = 2</td>
<td></td>
<td>c</td>
<td></td>
<td>b</td>
</tr>
<tr>
<td>3</td>
<td>d = 3</td>
<td></td>
<td>d</td>
<td></td>
<td>b, c</td>
</tr>
<tr>
<td>4</td>
<td>a = d</td>
<td></td>
<td>d</td>
<td>a</td>
<td>b, c, d</td>
</tr>
<tr>
<td>5</td>
<td>if a then</td>
<td>a</td>
<td></td>
<td></td>
<td>b, c, d</td>
</tr>
<tr>
<td>6</td>
<td>d = b + d</td>
<td>b, d</td>
<td>d</td>
<td>5</td>
<td>b, d</td>
</tr>
<tr>
<td>7</td>
<td>c = b + d</td>
<td>b, d</td>
<td>c</td>
<td>5</td>
<td>b, d</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>a = b + c</td>
<td>b, c</td>
<td>a</td>
<td></td>
<td>b, c</td>
</tr>
<tr>
<td>11</td>
<td>print a</td>
<td>a</td>
<td></td>
<td></td>
<td>a</td>
</tr>
</tbody>
</table>

We're not done yet! Remember slice for node 5 w.r.t. ref(5)!

slice currently contains: 10, 8, 7, 6, 5, 3, 2, 1

Step 2: relevant(11) = \{a\}

Step 3: relevant(10) = relevant(11) \setminus def(10) = \{a\} \setminus \{a\} = \{
relevant(10) = relevant(10) \cup ref(10) = \{\} \cup \{b, c\} = \{b, c\}
Since node 10 defines a variable relevant at node 11, it is included into the slice.

Step 3: relevant(9) = relevant(10) \setminus def(9) = \{b, c\} \setminus \{d\} = \{b, c\}

Step 3: relevant(8) = relevant(9) \setminus def(8) = \{b, c\} \setminus \{b\} = \{c\}

Step 3: relevant(8) = relevant(8) \cup ref(8) = \{c\} \cup \{b\} = \{b, c\}
Since node 8 defines a variable relevant at node 9, it is included into the slice. Since control(8) = 5, node 5 is included into the slice.
The slice for node 5 with respect to ref(5) is computed below.

Step 3: relevant(7) = relevant(10) \setminus def(7) = \{b, c\} \setminus \{c\} = \{b\}

Step 3: relevant(7) = relevant(7) \cup ref(7) = \{b\} \cup \{b, d\} = \{b, d\}
Since node 7 defines a variable relevant at node 10, it is included into the slice. Since control(7) = 5, node 5 is included into the slice.
The slice for node 5 with respect to ref(5) is computed below.

Step 3: relevant(6) = relevant(7) \setminus def(6) = \{b, d\} \setminus \{d\} = \{b\}

Step 3: relevant(6) = relevant(6) \cup ref(6) = \{b\} \cup \{b, d\} = \{b, d\}
Since node 6 defines a variable relevant at node 7, it is included into the slice.

Step 3: relevant(5) = relevant(6) \cup relevant(8) = \{b, d\} \cup \{b, c\} = \{b, c, d\}

Step 3: relevant(4) = relevant(5) \setminus def(4) = \{b, c, d\} \setminus \{a\} = \{b, c, d\}

Step 3: relevant(3) = relevant(4) \setminus def(3) = \{b, c, d\} \setminus \{d\} = \{b, c\}

Step 3: relevant(3) = relevant(3) \cup ref(3) = \{b, c\} \cup \{\} = \{b, c\}
Since node 3 defines a variable relevant at node 4, it is included into the slice.

Step 3: relevant(2) = relevant(3) \setminus def(2) = \{b, c\} \setminus \{c\} = \{b\}

Step 3: relevant(2) = relevant(2) \cup ref(2) = \{b\} \cup \{\} = \{b\}
Since node 2 defines a variable relevant at node 3, it is included into the slice.

Step 3: relevant(1) = relevant(2) \setminus def(1) = \{b\} \setminus \{\} = \{b\}

Step 3: relevant(1) = relevant(1) \cup ref(1) = \{\} \cup \{\} = \{\}
Since node 1 defines a variable relevant at node 2, it is included into the slice.
Let's take care of that subslice

<table>
<thead>
<tr>
<th>n</th>
<th>Statement</th>
<th>ref(n)</th>
<th>def(n)</th>
<th>control(n)</th>
<th>relevant(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>b = 1</td>
<td></td>
<td>b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>c = 2</td>
<td></td>
<td>c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>d = 3</td>
<td></td>
<td>d</td>
<td></td>
<td>{}</td>
</tr>
<tr>
<td>4</td>
<td>a = d</td>
<td>d</td>
<td>a</td>
<td></td>
<td>{d}</td>
</tr>
<tr>
<td>5</td>
<td>if a then</td>
<td>a</td>
<td></td>
<td></td>
<td>{a}</td>
</tr>
<tr>
<td>6</td>
<td>d = b + d</td>
<td>b, d</td>
<td>d</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>c = b + d</td>
<td>b, d</td>
<td>c</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>else</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>b = b + 1</td>
<td>b</td>
<td>b</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>9</td>
<td>d = b + 1</td>
<td>b</td>
<td>d</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>endif</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>a = b + c</td>
<td>b, c</td>
<td>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>print a</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Step 2: relevant(5) = {a}
Step 3: relevant(4) = relevant(5) - def(4) = {a} - {a} = {}
Since node 4 defines a variable relevant at node 5, it is included into the slice.
relevant(4) = relevant(4) ∪ ref(4) = {} ∪ {d} = {d}
Step 3: relevant(3) = relevant(4) - def(3) = {d} - {d} = {}
Since node 3 defines a variable relevant at node 4, it is included into the slice.
relevant(3) = relevant(3) ∪ ref(3) = {} ∪ {} = {}
Since the relevant set is empty, no more nodes will be included into the slice.

final slice contains: 10, 8, 7, 6, 5, 4, 3, 2, 1

slice for <5, {a}>
More reading

- The nice worked examples in these slides come from:
  - Program Slicing for Object-Oriented Programming Languages, Christoph Steindl (dissertation)
- If you want to dig in on these specific worked examples, take a look at Chapter 3 of the dissertation:
  - [http://www.ssw.uni-linz.ac.at/General/Staff/CS/Research/Publications/Ste99a.html](http://www.ssw.uni-linz.ac.at/General/Staff/CS/Research/Publications/Ste99a.html)
- A more comprehensive resource:
  - Cooper and Torczon's Engineering a Compiler textbook
  - [http://www.r-5.org/files/books/computers/compilers/writing/Keith_Cooper_Linda_Torczon-Engineering_a_Compiler-EN.pdf](http://www.r-5.org/files/books/computers/compilers/writing/Keith_Cooper_Linda_Torczon-Engineering_a_Compiler-EN.pdf)
What about loops?

- If we have loops, we have to keep iterating over the CFG until our slice and our relevant sets stabilize.
- You won’t be required to handle loops for your homework, but it’s pretty fun if you’re interested :)

Let’s do this!

- Fire up your

- This is going to be our last programming assignment of the semester, so get ready to do some language hacking :)