Slices

Chapter 10

Program slice

- Analyze program by focusing on parts of interest, disregarding uninteresting parts.
 - The point of slices is to separate a program into components that have a useful functional meaning
 - Ignore those parts that do not contribute to the functional meaning of interest
 - Cannot do this with du-paths, as slices are not simply sequences of statements or statement fragments



 A program slice is a set of program statements that contributes to or affects a value of a variable at some point in a program

Program slice – Formally

- Given a program P and a set of variables V in P, a slice on V at statement n, S(V, n), is the set of all statements and statement fragments in P prior to the node n that contribute to the values of variables in V at node n.
 - Usually statements and fragments correspond to numbered nodes in a program graph, so S(V, n) is a set of node numbers.
 - "Prior to" is a dynamic execution time notion

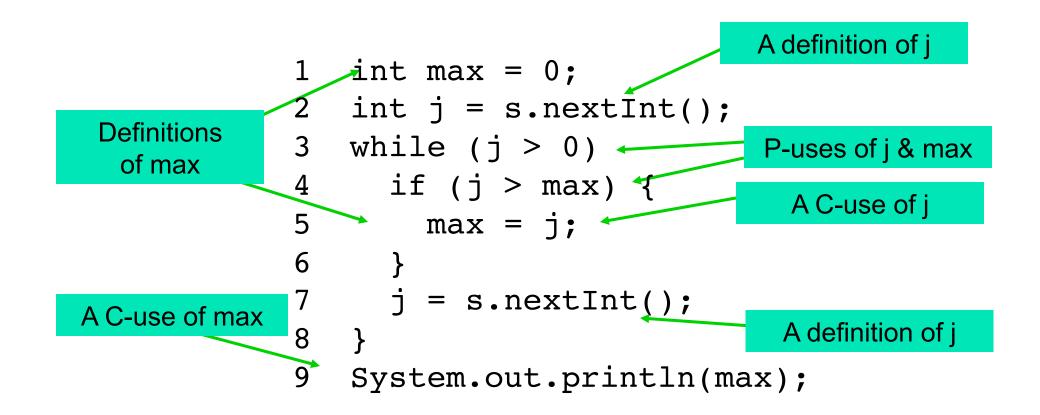
Program slice – meaning of "contributes to"

- Refine meaning of usage and defining nodes
 - P-use used in a decision predicate
 - C-use used in a computation
 - O-use used for output
 - L-use used for location (pointers, subscripts)
 - I-use used for iteration (loop counters, loop indices)
 - I-def defined by input
 - A-def defined by assignment
- Textbook excludes all non-executable statements such as variable declarations

Program slide – meaning of "contributes to" – 2

- What to include in S(V,n)? Consider a single variable v
 - Include all I-def, A-def
 - Include any C-use, P-use of v, if excluding it would change the value of v
 - Include any P-use or C-use of another variable, if excluding it would change the value of v
 - L-use and I-use
 - Inclusion is a judgment call, as such use does cause problems
 - Exclude all non-executable nodes such as variable declarations – if a slice is not to be compliable
 - Exclude O-use, as does not change the value of v





Example 1 -some slices

- This not an exciting program wrt to slices
 - S(max, 9) = { 1, 4, 5, 9 }
 - S (max, 9) = { 1, 2, 3, 4, 5, 6, 7, 8, 9 }
 - S (max, 5) = { 1, 4, 5, 6, 8 }
 - S(max, 5) = { 1, 2, 3, 4, 5, 6, 7, 8 }
 - S(j,7) = { 2, 3, 4, 56, 7, 8 }
 - S(j, 5) = {1, 2, 3, 4, 5, 6, 7, 8}

- Make slices on one variable
 - Sometimes slices with more variables are trivial super sets of a one variable case, then a slice on many variables is useful
 - Do not make a slice S(V, n) where the variables of interest are not in node n
 - Leads to slices that are too big

- Make slices for all A-def nodes
- Make slices for all P-def nodes very useful in decision intensive programs
- Try to make slices compliable
 - Means including declarations and compiler directives
 - Such slices become executable and more easily tested

- Avoid slices on C-use
 - They tend to be redundant
- Avoid slices on O-use
 - They are the union of A-def and I-def slices

- Relative complement of slices can have diagnostic value
 - If you have difficulty at a part, divide the program into two parts
 - If the error does not lie in one part, then it must be in the relative complement
- Slices contain define/reference information
 - When two slices are the same set, the corresponding paths are definition clear

- Slices and DD-paths have a many-to-many relationship
 - Nodes in one slice may be in many DD-paths, and nodes in one DD-path may be in many slices
 - Sometimes well-chosen relative complement slices can be identical to DD-paths
- Developing a lattice of slices can improve insight in potential trouble spots

Slices and programming practice

- Slice testing is an example where consideration of testing can lead to better program development
 - Build and test a program in slices
 - Merge / splice slices into larger programs
 - Use slice composition to re-develop difficult sections of program text