An Example of Dead Code Elimination Based on Program Dependence Graph

Zhiyuan Li

CS502

A Definition of Dead Code

- Dead code is the opposite of *useful* code
- Useful code consists of useful operations which are defined transitively as follows:
- (i) Any operation that *may* be executed and, if executed, will generate externally observable effect (e.g. writing to a file) is *useful*. (NOTE: we do not consider exceptions generated by software errors, such as divided by zero, as useful.
- (ii) Any operation on which a useful operation may depend is also useful

An Algorithm to Mark Useful Operations

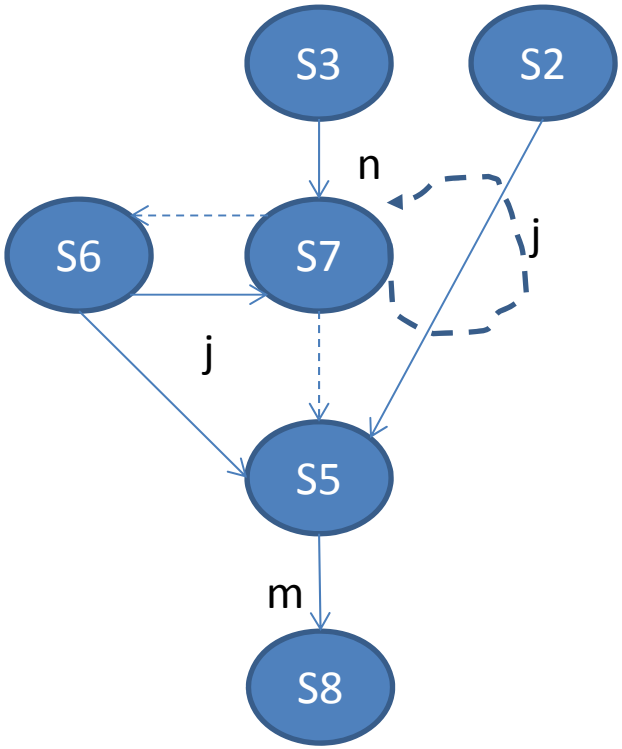
- The first step is to mark all reachable operations, i.e. operations that are reachable from the program (or the function, if we analyze intra-procedurally) entry point
  - This can be done simply by performing a DFS from the entry
- The second step is to traverse all reachable operations and mark those that generate externally observable operations (NOTE: we assume the compiler has access to the definition of all such operations.) Denote this set of useful operations, S_0.
- We build the program dependence graph and mark all operations that are reachable from operations in S_0, in reversed direction of the dependence edges. These operations, are the useful operations.

Notes

- We need to add the entry and the exit to the set of useful operations.
- Operations not marked as useful are dead and can be removed.
- In the first step of our algorithm, the reachability of operations can be *flow-condition insensitive* (i.e. not examining the branch conditions) or *flow-condition sensitive*
  - *The former may remove fewer dead operations than the latter*

A simple example

S1. i <- 1
S2. j <- 2
S3. n <- 4
S4. i <- i+j
S5. m <- j+1
S6. j <- j+2
S7. if j > n goto S4
S8. print(m)



NOTE:
After removing S4 Goto S4 must be changed to goto S5

Program Dependence Sub-Graph showing useful operations
Data dependence edge:
Control dependence edge: