RDA Write Up Draft

Reaching Definitions is a form of data-flow analysis concerned with statically and conservatively determining, for each statement in a program, which possible definitions can reach that statement. In this implementation of RDA, we are interested in determining the potential assignments that can influence the value of a variable at a particular statement (either upon entering that statement, or upon exiting of that statement) as well as determining what those values are (if possible).

For example, consider the simple program below.

Example 1 - $[y:=5]^1$; $[x:=y]^2$; $[y:=1]^3$

If we are interested in the potential values of y at statement 3, it is clear that upon entry of the statement: y's value has only been influenced by statement 1 (with a value of 5 in this case). Similarly, upon exit of the statement: y's value has been influence only by statement 3 (since it "overwrites" the assignment at statement 1). However, in more complex flow structures such as while loops, the list of potential assignments that can influence a variable's value at a given statement may be larger. Consider the more complex program below.

Example 2 - $[x:=5]^1$; $[y:=1]^2$; while $[x>1]^3$ do ($[y:=10]^4$; $[x:=x-1]^5$)

Say we are interested in determining the potential assignments that influence the value of variable x at the entry of statement 4. In this situation, we must consider the case where the program has entered the while loop for the first time (statement 1 is then responsible for the value) as well as the case in which the program is entering the statement on the 2^{nd} (or greater) iteration of the while loop (statement 5 is then responsible for the value). The Halting Problem tells us that it is impossible to determine whether or not a program will terminate prior to runtime, and as a result we do not know whether the program will enter the loop once, an infinite number of times, or at all. We therefore conservatively report all of these cases.

The ultimate result of this analysis is an RDAEntry/RDAExit list, which contains for each statement in the program the potential values for every variable in the scope of that statement. The first step in producing these entry/exit lists, is collecting every assignment in the program (the statement that it takes place, variable name, and value of assignment if possible). This task is performed by the AssignmentCollector program, which outputs the results in a csv file. In Example 2, AssignmentCollector would write {(x, 1, 5), (x, 5, ??), (y, 2, 1), (y, 4, 10)}, the format being {(variable, statement, value)}.

Parallel to the requirement of an assignment list is a control flow graph, which determines for each statement, the set of statements that can be entered from that statement. This is called the forward control flow, which can be used to calculate the reverse control flow (for each statement, the set of statements that can enter statement). The forward flow for Example 2 is: {(1, {2}), (2, {3}), (3, {4, END}), (4, {5}), (5, {3})}

The information returned by the AssignmentCollector is utilized to generate the Kill/Gen sets. The Kill set, for a given assignment statement L, is composed of every assignment to the variable in L in the entire program. Continuing with our analysis of Example 2, the Kill set for statement #4 is [(y,2), (y,4)] where 2 and 4 are the statements in which there is an assignment for y. The Gen set is composed of the single assignment that takes place in the statement of interest. In Example 2, the Gen set for statement 4 is {(y,4)}. In practice and in the example below, the values of the variables are reported rather than the statement of the assignment.

The RDAEntryExitList program utilizes the control flow graph and Kill/Gen sets to generate the Entry and Exit sets for each statement in the program. For the initial statement in the scope (the first line of a function, for example) the Entry set is given by the tuples of all variables that are initialized along with a question mark (to indicate an unknown value). In Example 2, the Entry set of statement 1 is: [(x,?), (y,?)]. The Entry set for statements that are not initial statements is given by the Exit set for every statement that could have entered the statement of interest. The Exit set for a given statement L is given by the Entry set of L first differenced with the Kill set associated with L, then taken in union with the Gen set associated with L. To make this concrete, we will work through the Entry and Exit sets of each statement in Example 2:

RDAEntry(1) = {(x,?), (y,?)}	#Initializing statement
RDAExit(1) = {(x,5), (y,?)}	#RDAEntry(1) / (x,?) <Kill> U (x,5) <Gen>
RDAEntry(2) = {(x,?), (y,?)}	#RDAExit(1)
RDAExit(2) = {(x,5), (y,?)}	#RDAEntry(2) / (y,?) U (y,1)
RDAEntry(3) = {(x,5), (y,1), (x,??), (y,10)}	#RDAExit(2) U RDAExit(5) since the while loop can be entered from 3 or 5 (second or more iterations
RDAExit(3) = {(x,5), (y,1), (x,??), (y,10)}	#RDAEntry(3)
RDAEntry(4) = {(x,5), (y,1), (x,??), (y,10)}	#RDAExit(3)
RDAExit(4) = {(x,5), (x,??), (y,10)}	#RDAEntry(4) / (y,1) U (y,10)
RDAEntry(5) = {(x,5), (x,??), (y,10)}	#RDAExit(4)
RDAExit(5) = {(x,??), (y,10)}	#RDAEntry(5) / (x,5) U (x,??)

*Note: the '??' appears because x is assigned an expression. It is generally not possible to know what expressions will evaluate to prior to runtime.

Notice that RDAEntry(3) is calculated using information from both RDAExit(2) and RDAExit(5). This means that an algorithm which works its way from top to bottom is insufficient. Rather, a worklist is used that interrupts the process and jumps to tasks that are required sooner. In this case, when the algorithm reaches RDAEntry(3), it notices that it requires information from RDAExit(5) and appends it to the worklist to be processed. This process can be visualized as the algorithm first working from top to bottom, then when needed jumping down and working back up. In some cases, a statement needs information from more than one point not yet reached. When this occurs, the algorithm must use chaotic iteration to randomly determine which of those statements to jump to first so that it does not repeatedly take paths that do not resolve the lack of information.

Once these sets are produced and written to file, they can be used to conservatively determine what values any variable can take at any point in the program. In the larger context of multilingual analysis, RDA may be used to check the possible values of an argument in a function call between various languages.