Lecture Notes: A Dataflow Analysis Framework for WHILE3ADDR

15-819O: Program Analysis
Jonathan Aldrich
jonathan.aldrich@cs.cmu.edu

Lecture 2

1 Defining a dataflow analysis

In order to make the definition of a dataflow analysis precise, we need to first examine how a dataflow analysis represents information about the program. The analysis will compute some dataflow information σ at each program point. Typically σ will tell us something about each variable in the program. For example, σ may be a mapping from variables to abstract values taken from some set L:

$$\sigma \in Var \to L$$

Here, L represents the set of abstract values we are interested in tracking in the analysis. This will vary from one analysis to another. Consider the example of *zero analysis*, in which we want to track whether each variable is zero or not. For this analysis L may represent the set $\{Z, N, ?\}$. Here the abstract value Z represents the value 0, N represents all nonzero values. We use $?$ for the situations when we do not know whether a variable is zero or not.

Conceptually, each abstract value is intended to represent a set of one or more concrete values that may occur when a program executes. In order to understand what an abstract value represents, we define an abstraction function α mapping each possible concrete value to an abstract value:

$$\alpha : \mathbb{Z} \to L$$

For zero analysis we simply define the function so that 0 maps to Z and all other integers map to N:

$$\begin{aligned}\alpha_Z(0) &= Z \\ \alpha_Z(n) &= N \quad \text{where } n \neq 0\end{aligned}$$

The core of any program analysis is how individual instructions in the program are analyzed. We define this using *flow functions* that map the dataflow information at the program point immediately before an instruction to the dataflow information after that instruction. A flow function should represent the semantics of the instruction, but do so abstractly in terms of the abstract values tracked by the analysis. We will describe more precisely what we mean by the semantics of the instruction when we talk about the correctness of dataflow analysis. As an example, though, we can define the flow functions f_Z for zero analysis as follows:

$$\begin{aligned}
f_Z[\![x := 0]\!](\sigma) &= [x \mapsto Z]\sigma \\
f_Z[\![x := n]\!](\sigma) &= [x \mapsto N]\sigma \qquad \text{where } n \neq 0 \\
f_Z[\![x := y]\!](\sigma) &= [x \mapsto \sigma(y)]\sigma \\
f_Z[\![x := y \ op \ z]\!](\sigma) &= [x \mapsto ?]\sigma \\
f_Z[\![\text{goto } n]\!](\sigma) &= \sigma \\
f_Z[\![\text{if } x = 0 \text{ goto } n]\!](\sigma) &= \sigma
\end{aligned}$$

The first flow function above is for assignment to a constant. In the notation, we represent the form of the instruction as an implicit argument to the function, which is followed by the explicit dataflow information argument, in the form $f_Z[\![I]\!](\sigma)$. If we assign 0 to a variable x, then we should update the input dataflow information σ so that x now maps to the abstract value Z. The notation $[x \mapsto Z]\sigma$ denotes dataflow information that is identical to σ except that the value in the mapping for x is updated to refer to Z.

The next flow function is for copies from a variable y to another variable x. In this case we just copy the dataflow information: we look up y in σ, written $\sigma(y)$, and update σ so that x maps to the same abstract value as y.

We define a generic flow function for arithmetic instructions. In general, an arithmetic instruction can return either a zero or a nonzero value, so we use the abstract value ? to represent our uncertainty. Of course, we could have written a more precise flow function here, which could return a more specific abstract value for certain instructions or operands. For example,

if the instruction is subtraction and the operands are the same, we know that the result is zero. Or, if the instruction is addition, and the analysis information tells us that one operand is zero, then we can deduce that the addition is really a copy and we can use a flow function similar to the copy instruction above. These examples could be written as follows (we would still need the generic case above for instructions that do not fit these special cases):

$$\begin{aligned} f_Z[\![x := y - y]\!](\sigma) &= [x \mapsto Z]\sigma \\ f_Z[\![x := y + z]\!](\sigma) &= [x \mapsto \sigma(y)]\sigma \quad \text{where } \sigma(z) = Z \end{aligned}$$

Exercise 1. Define another flow function for some arithmetic instruction and certain conditions where you can also provide a more precise result than $?$.

The flow function for conditional and unconditinal branches is trivial: the analysis result is unaffected by this instruction, which does not change the state of the machine other than to change the program counter.

We can provide a better flow function for conditional branches if we distinguish the analysis information produced when the branch is taken or not taken. To do this, we extend our notation once more in defining flow functions for branches, using a subscript to the instruction to indicate whether we are specifying the dataflow information for the case where the condition is true (T) or when it is false (F). For example, to define the flow function for the true condition when testing a variable for equality with zero, we use the notation $f_Z[\![\text{if } x = 0 \text{ goto } n]\!]_T(\sigma)$. In this case we know that x is zero so we can update σ with the Z lattice value. Conversely, in the false condition we know that x is nonzero:

$$\begin{aligned} f_Z[\![\text{if } x = 0 \text{ goto } n]\!]_T(\sigma) &= [x \mapsto Z]\sigma \\ f_Z[\![\text{if } x = 0 \text{ goto } n]\!]_F(\sigma) &= [x \mapsto N]\sigma \end{aligned}$$

Exercise 2. Define a flow function for a conditional branch testing whether a variable x is less than zero.

2 Running a dataflow analysis

The point of developing a dataflow analysis is to compute information about possible program states at each point in the program. For example, in the case of zero analysis, whenever we divide some expression by a variable x, we'd like to know whether x must be zero (represented by the abstract value Z) or may be zero (represented by ?) so that we can warn the developer of a divide by zero error.

Straightline code can be analyzed in a straightforward way, as one would expect. We simulate running the program in the analysis, using the flow function to compute, for each instruction in turn, the dataflow analysis information after the instruction from the information we had before the instruction. We can track the analysis information using a table with a column for each variable in the program and a row for each instruction, so that the information in a cell tells us the abstract value of the column's variable immediately after the row's instruction. For example, consider the program:

$$
\begin{array}{ll}
1: & x := 0 \\
2: & y := 1 \\
3: & z := y \\
4: & y := z + x \\
5: & x := y - z
\end{array}
$$

We would compute dataflow analysis information as follows:

	x	y	z
1	Z		
2	Z	N	
3	Z	N	N
4	Z	N	N
5	?	N	N

Notice that the analysis is imprecise at the end with respect to the value of x. We were able to keep track of which values are zero and nonzero quite well through instruction 4, using (in the last case) the flow function that knows that adding a variable which is known to be zero is equivalent to a copy. However, at instruction 5, the analysis does not know that y and z are equal, and so it cannot determine whether x will be zero or not. Because the analysis is not tracking the exact values of variables, but rather approximations, it will inevitably be imprecise in certain situations. How-

ever, in practice well-designed approximations can often allow dataflow analysis to compute quite useful information.

3 Alternative paths and dataflow joins

Things are more interesting in WHILE3ADDR code that represents an if statement. In this case, there are two possible paths through the program. Consider the following simple example:

$$
\begin{array}{ll}
1: & \text{if } x = 0 \text{ goto } 4 \\
2: & y := 0 \\
3: & \text{goto } 6 \\
4: & y := 1 \\
5: & x := 1 \\
6: & z := y
\end{array}
$$

We could begin by analyzing one path through the program, for example the path in which the branch is not taken:

	x	y	z
1	Z_T, N_F		
2	N	Z	
3	N	Z	
4			
5			
6	N	Z	Z

In the table above, the entry for x on line 1 indicates the different abstract values produced for the true and false conditions of the branch. We use the false condition (x is nonzero) in analyzing instruction 2. Execution proceeds through instruction 3, at which point we jump to instruction 6. The entries for lines 4 and 5 are blank because we have not analyzed a path through the program that executes this line.

A side issue that comes up when analyzing instruction 1 is what should we assume about the value of x? In this example we will assume that x is an input variable, because it is used before it is defined. For input variables, we should start the beginning of the program with some reasonable assumption. If we do not know anything about the value x can be, the best choice is to assume it can be anything. That is, in the initial environment σ_0, x is mapped to $?$.

We turn now to an even more interesting question: how to consider the alternative path through this program in our analysis. The first step is to analyze instructions 4 and 5 as if we had taken the true branch at instruction 1. Adding this, along with an assumption about the initial value of x at the beginning of the program (which we will assume is line 0), we have:

	x	y	z	
0	?			
1	Z_T, N_F			
2	N	Z		
3	N	Z		
4	Z	N		
5	N	N		
6	N	Z	Z	*note: incorrect!*

Now we have a dilemma in analyzing instruction 6. We already analyzed it with respect to the previous path, assuming the dataflow analysis we computed from instruction 3, where x was nonzero and y was zero. However, we now have conflicting information from instruction 5: in this case, x is also nonzero, but y is nonzero in this case. Therefore, the results we previously computed for instruction 6 are invalid for the path that goes through instruction 4.

A simple and safe resolution of this dilemma is simply to choose an abstract value for x and y that combine the abstract values computed along the two paths. The incoming abstract values for y are N and Z, which tells us that y may be either nonzero or zero. We can represent this with the abstract value $?$ indicating that we do know know if y is zero or not at this instruction, because of the uncertainty about how we reached this program location. We can apply similar logic in the case of x, but because x is nonzero on both incoming paths we can maintain our knowledge that x is nonzero. Thus, we should reanalyze instruction 5 assuming the dataflow analysis information $\{x \mapsto N, y \mapsto ?\}$. The results of our final analysis are shown below:

	x	y	z	
0	?			
1	Z_T, N_F			
2	N	Z		
3	N	Z		
4	Z	N		
5	N	N		
6	N	?	?	*corrected*

We can generalize the procedure of combining analysis results along multiple paths by using a join operation, $\sqcup$. The idea is that when taking two abstract values $l_1, l_2 \in L$, the result of $l_1 \sqcup l_2$ is always an abstract value l_j that generalizes both l_1 and l_2.

In order to define what generalizes means, we can define a partial order $\sqsubseteq$ over abstract values, and say that l_1 and l_2 are at least as precise as l_j, written $l_1 \sqsubseteq l_j$. Recall that a partial order is any relation that is:

- reflexive: $\forall l : l \sqsubseteq l$
- transitive: $\forall l_1, l_2, l_3 : l_1 \sqsubseteq l_2 \wedge l_2 \sqsubseteq l_3 \Rightarrow l_1 \sqsubseteq l_3$
- anti-symmetric: $\forall l_1, l_2 : l_1 \sqsubseteq l_2 \wedge l_2 \sqsubseteq l_1 \Rightarrow l_1 = l_2$

A set of values L that is equipped with a partial order $\sqsubseteq$, and for which the least upper bound of any two values in that ordering $l_1 \sqcup l_2$ is unique and is also in L, is called a *join-semilattice*. Any join-semilattice has a maximal element $\top$. We will require that the abstract values used in dataflow analyses form a join-semilattice. We will use the term lattice for short; as we will see below, this is the correct terminology for most dataflow analyses.

For zero analysis, we define the partial order with $Z \sqsubseteq ?$ and $N \sqsubseteq ?$, where $Z \sqcup N = ?$ and the $\top$ lattice element is $?$. In order to emphasize the lattice concept, we will use $\top$ in place of $?$ for zero analysis in the following notes.

We have now considered all the elements necessary to define a dataflow analysis. These are:

- a lattice $(L, \sqsubseteq)$
- an abstraction function α
- initial dataflow analysis assumptions σ_0
- a flow function f

Note that based on the theory of lattices, we can now propose a generic natural default for the initial dataflow information: a σ that maps each variable that is in scope at the beginning of the program to $\top$, indicating uncertainty as to that variable's value.

4 Dataflow analysis of loops

We now consider WHILE3ADDR programs that represent looping control flow. While an if statement produces two alternative paths that diverge and later join, a loop produces an potentially unbounded number of paths into the program. Despite this, we would like to analyze looping programs in bounded time. Let us examine how through the following simple looping example:

$$
\begin{array}{ll}
1: & x := 10 \\
2: & y := 0 \\
3: & z := 0 \\
4: & \text{if } x = 0 \text{ goto } 8 \\
5: & y := 1 \\
6: & x := x - 1 \\
7: & \text{goto } 4 \\
8: & x := y
\end{array}
$$

Let us first consider a straight-line analysis of the program path that enters the loop and runs through it once:

	x	y	z
1	N		
2	N	Z	
3	N	Z	Z
4	Z_T, N_F	Z	Z
5	N	N	Z
6	$\top$	N	Z
7	$\top$	N	Z
8			

So far things are straightforward. We must now analyze instruction 4 again. This situation should not be surprising however; it is analogous to the situation when merging paths after an if instruction. To determine the analysis information at instruction 4, we should join the dataflow analysis

information flowing in from instruction 3 with the dataflow analysis information flowing in from instruction 7. For x we have $N \sqcup \top = N$. For y we have $Z \sqcup N = \top$. For z we have $Z \sqcup Z = Z$. The information coming out of instruction 4 is therefore the same as before, except that for y we now have $\top$.

We can now choose between two paths once again: staying within the loop or exiting out to instruction 8. We will choose (arbitrarily for now) to stay within the loop and consider instruction 5. This is our second visit to instruction 5 and we have new information to consider: in particular, since we have gone through the loop, the assignment $y := 1$ has been executed and we have to assume that y may be nonzero coming into instruction 5. This is accounted for by the latest update to instruction 4's analysis information, in which y is mapped to $\top$. Thus the information for instruction 4 describes both possible paths. We must update the analysis information for instruction 5 so it does so as well. In this case, however, the instruction assigns 1 to y, so we still know that y is nonzero after the instruction executes. In fact, analysing the instruction again with the updated input data does not change the analysis results for this instruction.

A quick check shows that going through the remaining instructions in the loop, and even coming back to instruction 4, the analysis information will not change. That is because the flow functions are deterministic: given the same input analysis information and the same instruction, they will produce the same output analysis information. If we analyze instruction 6, for example, the input analysis information from instruction 5 is the same input analysis information we used when analyzing instruction 6 the last time aroudn. Thus instruction 6's output information will not change, so therefore instruction 7's input information will not change, and so on. No matter which instruction we run the analysis on, anywhere in the loop (and in fact before the loop), the analysis information will not change.

We say that the dataflow analysis has reached a *fixed point*.[1] In mathematics, a fixed point of a function is a data value v that is mapped to itself by the function, i.e. $f(v) = v$. In this situation, the mathematical function is the flow function, and the fixed point is a tuple of the dataflow analysis values at each point in the program (up to and including the loop). If we invoke the flow function on the fixed point, we get the same fixed point back.

Once we have reached a fixed point of the function for this loop, it is clear that further analysis of the loop will not be useful. Therefore, we will

[1] sometimes abbreviated fixpoint

proceed to analyze statement 8. The final analysis results are as follows:

	x	y	z	
1	N			
2	N	Z		
3	N	Z	Z	
4	Z_T, N_F	$\top$	Z	*updated*
5	N	N	Z	*already at fixed point*
6	$\top$	N	Z	*already at fixed point*
7	$\top$	N	Z	*already at fixed point*
8	Z	$\top$	Z	

Quickly simulating a run of the program program shows that these results correctly approximate actual execution. The uncertainty in the value of x at instructions 6 and 7 is real: x is nonzero after these instructions, except the last time through the loop, when it is zero. The uncertainty in the value of y at the end shows imprecision in the analysis; in this program, the loop always executes at least once, so y will be nonzero. However, the analysis (as currently formulated) cannot tell that the loop is executed even once, so it reports that it cannot tell if y is zero or not. This report is safe—it is always correct to say the analysis is uncertain—but not as precise as one might like.

The benefit of analysis, however, is that we can gain correct information about all possible executions of the program with only a finite amount of work. For example, in this case we only had to analyze the loop statements at most twice before recognizing that we had reached a fixed point. Since the actual program runs the loop 10 times—and could run many more times, if we initialized x to a higher value—this is a significant benefit. We have sacrificed some precision in exchange for coverage of all possible executions, a classic tradeoff in static analysis.

How can we be confident that the results of the analysis are correct, besides simulating every possible run of the program? After all, there may be many such runs in more complicated programs, for example when the behavior of the program depends on input data. The intuition behind correctness is the invariant that at each program point, the analysis results approximate all the possible program values that could exist at that point. If the analysis information at the beginning of the program correctly approximates the program arguments, then the invariant is true at the beginning of program execution. One can then make an inductive argument that the invariant is preserved as the program executes. In particular, when the pro-

gram executes an instruction, the instruction modifies the program's state. As long as the flow functions account for every possible way that instruction can modify the program's state, then at the analysis fixed point they will have produced a correct approximation of the actual program's execution after that instruction. We will make this argument more precise in a future lecture.

5 A convenience: the $\bot$ abstract value and complete lattices

As we think about defining an algorithm for dataflow analysis more precisely, a natural question comes up concerning how instruction 4 is analyzed in the example above. On the first pass we analyzed it using the dataflow information from instruction 3, but on the second pass we had to consider dataflow information from instruction 3 as well as from instruction 7.

It would be more consistent if we could just say that analyzing instruction 4 always uses the incoming dataflow analysis information from all instructions that could precede it in execution. That way we do not have to worry about following a specific path during analysis. Doing this requires having a dataflow value from instruction 7, however, even if instruction 7 has not yet been analyzed. We could do this if we had a dataflow value that is always ignored when it is joined with any other dataflow value. In other words, we need a abstract dataflow value $\bot$ such that $\bot \sqcup l = l$.

We name this abstract value $\bot$ because it plays a dual role to the value $\top$: it sits at the bottom of the dataflow value lattice. While $\top \sqcup l = \top$, we have $\bot \sqcup l = l$. For all l we have the identity $l \sqsubseteq \top$ and correspondingly $\bot \sqsubseteq l$. There is an greatest lower bound operator *meet*, $\sqcap$, which is dual to $\sqcup$, and the meet of all dataflow values is $\bot$.

A set of values L that is equipped with a partial order $\sqsubseteq$, and for which both least upper bounds $\sqcup$ and greatest lower bounds $\sqcap$ exist in L and are unique, is called a (complete) *lattice*.

The theory of $\bot$ and complete lattices provides an elegant solution to the problem mentioned above. We can initialize every dataflow value in the program, except at program entry, to $\bot$, indicating that the instruction there has not yet been analyzed. We can then always merge all input values to a node, whether or not the sources of those inputs have been analysed, because we know that any $\bot$ values from unanalyzed sources will simply be ignored by the join operator $\sqcup$.

6 Analysis execution strategy

The informal execution strategy outlined above operations by considering all paths through the program, continuing until the dataflow analysis information reaches a fixed point. This strategy can in fact be simplified. The argument for correctness outlined above, implies that for correct flow functions, it doesn't matter how we get to the mathematical fixed point of the analysis. This seems sensible: it would be surprising if the correctness of the analysis depended on which branch of an if statement we explore first. It is in fact possible to run the analysis on program instructions in any order we choose. As long as we continue doing so until the analysis reaches a fixed point, the final result will be correct. The simplest correct algorithm for executing dataflow analysis can therefore be stated as follows:

```
for Instruction i in program
    input[i] = ⊥
input[firstInstruction] = initialDataflowInformation

while not at fixed point
    pick an instruction i in program
    output = flow(i, input[i])
    for Instruction j in sucessors(i)
        input[j] = input[j] ⊔ output
```

Although in the previous presentation we have been tracking the analysis information immediately after each instruction, it is more convenient when writing down the algorithm to track the analysis information immediately before each instruction. This avoids the need for a distinguished location before the program starts.

In the code above, the termination condition is expressed abstractly. It can easily be checked, however, by running the flow function on each instruction in the program. If the results of analysis do not change as a result of analyzing any instruction, then the analysis has reached a fixed point.

How do we know the algorithm will terminate? The intuition is as follows. We rely on the choice of an instruction to be fair, so that each instruction is eventually considered. As long as the analysis is not at a fixed point, some instruction can be analyzed to produce new analysis results. If our flow functions are well-behaved (technically, if they are monotone, as discussed in a future lecture) then each time the flow function runs on a given instruction, either the results do not change, or they get more approximate (i.e. they are higher in the lattice). The intuition is that later runs of the flow

function consider more possible paths through the program and therefore produce a more approximate result which considers all these possibilities. If the lattice is of finite height—meaning there are at most a finite number of steps from any place in the lattice going up towards the $\top$ value—then this process must terminate eventually, because the analysis information cannot get any higher.

Although the simple algorithm above always terminates and results in the correct answer, it is not always the most efficient. Typically, for example, it is beneficial to analyze the program instructions in order, so that results from earlier instructions can be used to update the results of later instructions. It is also useful to keep track of a list of instructions for which there has been a change, since the instruction was last analyzed, in the result dataflow information of some predecessor instruction. Only those instructions need be analyzed; reanalyzing other instructions is useless since the input has not changed and they will produce the same result as before. Kildall captured this intuition with his worklist algorithm, described in pseudocode below:

```
for Instruction i in program
    input[i] = ⊥
input[firstInstruction] = initialDataflowInformation
worklist = { firstInstruction }

while worklist is not empty
    take an instruction i off the worklist
    output = flow(i, input[i])
        for Instruction j in successors(i)
                if output ⋢ input[j]
                        input[j] = input[j] ⊔ output
                        add j to worklist
```

The algorithm above is very close to the generic algorithm declared before, except for the worklist that is used to choose the next instruction to analyze and to determine when a fixed point has been reached.

We can reason about the performance of this algorithm as follows. We only add an instruction to the worklist whenever the input data to some node changes, and the input for a given node can only change h times, where h is the height of the lattice. Thus we add at most $n * h$ nodes to the worklist, where n is the number of instructions in the program. After running the flow function for a node, however, we must test all its successors to find out if their input has changed. This test is done once for each

edge, for each time that the source node of the edge is added to the worklist: thus at most $e * h$ times, where e is the number of control flow edges in the successor graph between instructions. If each operation (such as a flow function, $\sqcup$, or $\sqsubseteq$ test) has cost $O(c)$, then the overall cost is $O(c*(n+e)*h)$, or $O(c * e * h)$ because n is bounded by e.

The algorithm above is still abstract in that we have not defined the operations to add and remove instructions from the worklist. We would like add to work list a set addition operation, so that no instruction appears in the worklist multiple times. The justification is that if we have just analysed the program with respect to an instruction, analyzing it again will not produce different results.

That leaves a choice of which instruction to remove from the worklist. We could choose among several policies, including last-in-first-out (LIFO) order or first-in-first-out (FIFO) order. In practice, the most efficient approach is to identify the strongly-connected components (i.e. loops) in the control flow graph of components and process them in topological order, so that loops that are nested, or appear in program order first, are solved before later loops. This works well because we do not want to do a lot of work bringing a loop late in the program to a fixed point, then have to redo this work when dataflow information from an earlier loop changes.

Within each loop, the instructions should be processed in reverse postorder. Reverse postorder is defined as the reverse of the order in which each node is last visited when traversing a tree. Consider the example from section **??** above, in which instruction 1 is an if test, instructions 2-3 are the then branch, instructions 4-5 are the else branch, and instruction 6 comes after the if statement. A tree traversal might go as follows: 1, 2, 3, 6, 3 (again), 2 (again), 1 (again), 4, 5, 4 (again), 1 (again). Some instructions in the tree are visited multiple times: once going down, once between visiting the children, and once coming up. The postorder, or order of the last visits to each node, is 6, 3, 2, 5, 4, 1. The reverse postorder is the reverse of this: 1, 4, 5, 2, 3, 6. Now we can see why reverse postorder works well: we explore both branches of the if statement (4-5 and 2-3) before we explore node 6. This ensures that, in this loop-free code, we do not have to reanalyze node 6 after one of its inputs changes.

Although analyzing code using the strongly-connected component and reverse postorder hueristics improves performance substantially in practice, it does not change the worst-case performance results described above.