Lecture Notes: The WHILE Language and WHILE3ADDR Representation

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

Lecture 2

1 The WHILE Language

In this course, we will study the theory of analyses using a simple programming language called WHILE, along with various extensions. The WHILE language is at least as old as Hoare's 1969 paper on a logic for proving program properties (to be discussed in a later lecture). It is a simple imperative language, with assignment to local variables, if statements, while loops, and simple integer and boolean expressions.

We will use the following metavariables to describe several different categories of syntax. The letter on the left will be used as a variable representing a piece of a program, while on the right, we describe the kind of program piece that variable represents:

S	statements
a	arithmetic expressions
x, y	program variables
n	number literals
P	boolean predicates

The syntax of WHILE is shown below. Statements S can be an assignment $x := a$, a skip statement which does nothing (similar to a lone semicolon or open/close bracket in C or Java), and if and while statements whose condition is a boolean predicate P. Arithmetic expressions a include variables x, numbers n, and one of several arithmetic operators, abstractly

represented by op_a. Boolean expressions include true, false, the negation of another boolean expression, boolean operators op_b applied to other boolean expressions, and relational operators op_r applied to arithmetic expressions.

$$
\begin{array}{lcl}
S & ::= & x := a \\
 & | & \text{skip} \\
 & | & S_1;\ S_2 \\
 & | & \text{if } P \text{ then } S_1 \text{ else } S_2 \\
 & | & \text{while } P \text{ do } S \\
a & ::= & x \\
 & | & n \\
 & | & a_1\ op_a\ a_2 \\
op_a & ::= & + \mid - \mid * \mid / \\
P & ::= & \text{true} \\
 & | & \text{false} \\
 & | & \text{not } P \\
 & | & P_1\ op_b\ P_2 \\
 & | & a_1\ op_r\ a_2 \\
op_b & ::= & \text{and} \mid \text{or} \\
op_r & ::= & < \mid \leq \mid = \mid > \mid \geq
\end{array}
$$

2 WHILE3ADDR: A Representation for Analysis

We could define the semantics of WHILE directly—and indeed we will do so when studying Hoare Logic.[1] For program analysis, however, the source-like definition of WHILE is somewhat inconvenient. Even a simple language such as WHILE can be complex to define. For example, WHILE has three separate syntactic forms—statements, arithmetic expressions, and boolean predicates—and we would have to define the semantics of each separately. A simpler and more regular representation of programs will help make our formalism simpler.

As a starting point, we will eliminate recursive arithmetic and boolean expressions and replace them with simple atomic statement forms, which are called instructions after the assembly language instructions that they resemble. For example, an assignment statement of the form $w = x * y + z$

[1] The supplemental Nielson et al. text also defines the semantics of the WHILE language given here.

will be rewritten as a multiply instruction followed by an add instruction. The multiply assigns to a temporary variable t_1, which is then used in the subsequent add:

$$t_1 = x * y$$
$$w = t_1 + z$$

As the translation from expressions to instructions suggests, program analysis is typically studied using a representation of programs that is not only simpler, but also lower-level than the source WHILE language. Typically high-level languages come with features that are numerous and complex, but can be reduced into a smaller set of simpler primitives. Working at the lower level of abstraction thus also supports simplicity in the compiler.

Control flow constructs such as if and while are similarly translated into simpler goto and conditional branch constructs that jump to a particular (numbered) instruction. For example, a statement of the form if P then S_1 else S_2 would be translated into:

$1:$ if P then goto 4
$2:$ S_2
$3:$ goto 5
$4:$ S_1
$5:$ *rest of program...*

The translation of a statement of the form while P do S is similar:

$1:$ if not P goto 4
$2:$ S
$3:$ goto 1
$4:$ *rest of program...*

This form of code is often called 3-address code, because every instruction is of a simple form with at most two source operands and one result operand. We now define the syntax for 3-address code produce from the WHILE language, which we will call WHILE3ADDR. This language consists of a set of simple instructions that load a constant into a variable, copy from one variable to another, compute the value of a variable from two others, or jump (possibly conditionally) to a new address n. A program is just a map from addresses to instructions:

$$
\begin{array}{lcl}
I & ::= & x := n \\
 & | & x := y \\
 & | & x := y \ op \ z \\
 & | & \text{goto } n \\
 & | & \text{if } x \ op_r \ 0 \text{ goto } n \\
op & ::= & + \mid - \mid * \mid / \\
op_r & ::= & < \mid = \\
P & \in & \mathbb{N} \rightharpoonup I
\end{array}
$$

Formally defining a translation from a source language such as WHILE to a lower-level intermediate language such as WHILE3ADDR is possible, but it is more appropriate for the scope of a compilers course. For the purposes of this course, the examples above should suffice as intuition. We will proceed by first formalizing program analysis in WHILE3ADDR, then having a closer look at its semantics in order to verify the correctness of the program analysis.

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.

Lecture Notes:
Dataflow Analysis Examples

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

Lecture 3

1 Constant Propagation

While zero analysis was useful for simply tracking whether a given variable is zero or not, constant propagation analysis attempts to track the constant values of variables in the program, where possible. Constant propagation has long been used in compiler optimization passes in order to turn variable reads and computations into constants, where possible. However, it is generally useful for analysis for program correctness as well: any client analysis that benefits from knowing program values (e.g. an array bounds analysis) can leverage it.

For constant propagation, we want to track what is the constant value, if any, of each program variable. Therefore we will use a lattice where the set L_{CP} is $\mathbb{Z} \cup \{\top, \bot\}$. The partial order is $\forall l \in L_{CP} : \bot \sqsubseteq l \wedge l \sqsubseteq \top$. In other words, $\bot$ is below every lattice element and $\top$ is above every element, but otherwise lattice elements are incomparable.

In the above lattice, as well as our earlier discussion of the lattice for zero analysis, we considered a lattice to describe individual variable values. We can lift the notion of a lattice to cover all the dataflow informition available at a program point. This is called a *tuple lattice*, where there is an element of the tuple for each of the variables in the program. For constant propagation, the elements of the set σ are maps from *Var* to L_{CP}, and the other operators and $\top/\bot$ are lifted as follows:

$$
\begin{array}{rcl}
\sigma & \in & \mathit{Var} \to L_{CP} \\
\sigma_1 \sqsubseteq_{lift} \sigma_2 & \mathit{iff} & \forall x \in \mathit{Var} : \sigma_1(x) \sqsubseteq \sigma_2(x) \\
\sigma_1 \sqcup_{lift} \sigma_2 & = & \{x \mapsto \sigma_1(x) \sqcup \sigma_2(x) \mid x \in \mathit{Var}\} \\
\top_{lift} & = & \{x \mapsto \top \mid x \in \mathit{Var}\} \\
\bot_{lift} & = & \{x \mapsto \bot \mid x \in \mathit{Var}\}
\end{array}
$$

We can likewise define an abstraction function for constant propagation, as well as a lifted version that accepts an environment E mapping variables to concrete values. We also define the initial analysis information to conservatively assume that initial variable values are unknown. Note that in a language that initializes all variables to zero, we could make more precise initial dataflow assumptions, such as $\{x \mapsto 0 \mid x \in \mathit{Var}\}$:

$$
\begin{array}{rcl}
\alpha_{CP}(n) & = & n \\
\alpha_{lift}(E) & = & \{x \mapsto \alpha_{CP}(E(x)) \mid x \in \mathit{Var}\} \\
\sigma_0 & = & \top_{lift}
\end{array}
$$

We can now define flow functions for constant propagation:

$$
\begin{array}{lll}
f_{CP}[\![x := n]\!](\sigma) & = [x \mapsto n]\sigma & \\
f_{CP}[\![x := y]\!](\sigma) & = [x \mapsto \sigma(y)]\sigma & \\
f_{CP}[\![x := y \mathit{\ op\ } z]\!](\sigma) & = [x \mapsto \sigma(y) \mathit{\ op}_{lift}\ \sigma(z)]\sigma & \\
 & \quad \text{where } n \mathit{\ op}_{lift}\ m = n \mathit{\ op\ } m & \\
 & \quad \text{and } n \mathit{\ op}_{lift}\ \top = \top & \text{(and symmetric)} \\
 & \quad \text{and } n \mathit{\ op}_{lift}\ \bot = n & \text{(and symmetric)} \\
f_{CP}[\![\text{goto } n]\!](\sigma) & = \sigma & \\
f_{CP}[\![\text{if } x = 0 \text{ goto } n]\!]_T(\sigma) & = [x \mapsto 0]\sigma & \\
f_{CP}[\![\text{if } x = 0 \text{ goto } n]\!]_F(\sigma) & = \sigma & \\
f_{CP}[\![\text{if } x < 0 \text{ goto } n]\!](\sigma) & = \sigma &
\end{array}
$$

We can now look at an example of constant propagation:

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

We would compute dataflow analysis information as follows. In this table we will use track the worklist and show updates using additional rows to show the operation of the algorithm:

stmt	worklist	x	y	z	w
0	1	$\top$	$\top$	$\top$	$\top$
1	2	3	$\top$	$\top$	$\top$
2	3	3	10	$\top$	$\top$
3	4,6	3	10	$0_T, \top_F$	$\top$
4	5,6	3	10	5	$\top$
5	6,7	3	10	5	$\top$
6	7	3	10	5	$\top$
7	$\varnothing$	3	10	5	3

2 Reaching Definitions

Reaching definitions analysis determines, for each use of a variable, which assignments to that variable might have set the value seen at that use. Consider the following program:

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

In this example, definitions 1 and 5 reach the use of y at 4.

Exercise 1. Which definitions reach the use of z at statement 4?

Reaching definitions can be used as a simpler but less precise version of constant propagation, zero analysis, etc. where instead of tracking actual constant values we just look up the reaching definition and see if it is a constant. We can also use reaching definitions to identify uses of undefined variables, e.g. if no definition from the program reaches a use.

For reaching definitions, we will use a new kind of lattice: a *set lattice*. Here, a dataflow lattice element will be the set of definitions that reach the current program point. Assume that DEFS is the set of all definitions in

the program. The set of elements in the lattice is the set of all subsets of DEFS—that is, the powerset of DEFS, written $\mathcal{P}^{\mathsf{DEFS}}$.

What should $\sqsubseteq$ be for reaching definitions? The intuition is that our analysis is more precise the *smaller* the set of definitions it computes at a given program point. This is because we want to know, as precisely as possible, where the values at a program point came from. So $\sqsubseteq$ should be the subset relation $\subseteq$: a subset is more precise than its superset. This naturally implies that $\sqcup$ should be $union$, and that $\top$ and $\bot$ should be the universal set DEFS and the empty set $\varnothing$, respectively.

In summary, we can formally define our lattice and initial dataflow information as follows:

$$\begin{array}{rcl} \sigma & \in & \mathcal{P}^{\mathsf{DEFS}} \\ \sigma_1 \sqsubseteq \sigma_2 & \textit{iff} & \sigma_1 \subseteq \sigma_2 \\ \sigma_1 \sqcup \sigma_2 & = & \sigma_1 \cup \sigma_2 \\ \top & = & \mathsf{DEFS} \\ \bot & = & \varnothing \\ \sigma_0 & = & \varnothing \end{array}$$

Instead of using the empty set for σ_0, we could use an artificial reaching definition for each program variable (e.g. x_0 as an artificial reaching definition for x) to denote that the variable is either uninitialized, or was passed in as a parameter. This is convenient if it is useful to track whether a variable might be uninitialized at a use, or if we want to consider a parameter to be a definition.

We will now define flow functions for reaching definitions. Notationally, we will write x_n to denote a definition of the variable x at the program instruction numbered n. Since our lattice is a set, we can reason about changes to it in terms of elements that are added (called GEN) and elements that are removed (called KILL) for each statement. This GEN/KILL pattern is common to many dataflow analyses. The flow functions can be formally defined as follows:

$$\begin{array}{lll} f_{RD}[\![I]\!](\sigma) & = \sigma - KILL_{RD}[\![I]\!] \cup GEN_{RD}[\![I]\!] & \\ \text{KILL}_{RD}[\![n : x := ...]\!] & = \{x_m | x_m \in \mathsf{DEFS}(x)\} & \\ \text{KILL}_{RD}[\![I]\!] & = \varnothing & \text{if } I \text{ is not an assignment} \\ \text{GEN}_{RD}[\![n : x := ...]\!] & = \{x_n\} & \\ \text{GEN}_{RD}[\![I]\!] & = \varnothing & \text{if } I \text{ is not an assignment} \end{array}$$

We would compute dataflow analysis information for the program shown above as follows:

stmt	worklist	defs
0	1	$\varnothing$
1	2	$\{y_1\}$
2	3	$\{y_1, z_1\}$
3	4,7	$\{y_1, z_1\}$
4	5,7	$\{y_1, z_4\}$
5	6,7	$\{y_5, z_4\}$
6	3,7	$\{y_5, z_4\}$
3	4,7	$\{y_1, y_5, z_1, z_4\}$
4	5,7	$\{y_1, y_5, z_4\}$
5	7	$\{y_5, z_4\}$
7	$\varnothing$	$\{y_7, z_1, z_4\}$

3 Live Variables

Live variable analysis determines, for each program point, which variables might be used again before they are redefined. Consider again the following program:

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

In this example, afger instruction 1, y is live, but x and z are not. Live variables analysis typically requires knowing what variable holds the main result(s) computed by the program. In the program above, suppose z is the result of the program. Then at the end of the program, only z is live.

Live variable analysis was originally developed for optimizatoin purposes: if a variable is not live after it is defined, we can remove the definition instruction. For example, instruction 7 in the code above could be optimized away, under our assumption that z is the only program result of interest.

We must be careful of the side effects of a statement, of course. Assigning a variable that is no longer live to null could have the beneficial side effect of allowing the garbage collector to collect memory that is no longer reachable—unless the GC itself takes into consideration which variables are live. Sometimes warning the user that an assignment has no effect can be useful for software engineering purposes, even if the assignment cannot safely be optimized away. For example, eBay found that FindBugs's analysis detecting assignments to dead variables was useful for identifying unnecessary database calls[1].

For live variable analysis, we will use a set lattice to track the set of live variables at each program point. The lattice is similar to that for reaching definitions:

$$\begin{array}{rcl} \sigma & \in & \mathcal{P}^{\mathsf{Var}} \\ \sigma_1 \sqsubseteq \sigma_2 & \textit{iff} & \sigma_1 \subseteq \sigma_2 \\ \sigma_1 \sqcup \sigma_2 & = & \sigma_1 \cup \sigma_2 \\ \top & = & \mathsf{Var} \\ \bot & = & \varnothing \end{array}$$

What is the initial dataflow information? This is a tricky question. To determine the variables that are live at the start of the program, we must reason about how the program will execute. But this is in fact the purpose of dataflow analysis. On the other hand, it is quite clear which variables are live at the *end* of the program: just the variable(s) holding the program result.

Consider how we might use this information to compute other live variables. Suppose the last statement in the program assigns the program result z, computing it based on some other variable x. Intuitively, that statement should make x live immediately above that statement, as it is needed to compute the program result z—but z should now no longer be live. We can use similar logic for the second-to-last statement, and so on. In fact, we can see that live variable analysis is a *backwards analysis*: we start with dataflow information at the *end* of the program and use flow functions to compute dataflow information at earlier statements.

Thus, for our "initial" dataflow information—and note that "initial" means the beginning of the program analysis, but the end of the program—we have:

[1] see Ciera Jaspan, I-Chin Chen, and Anoop Sharma, *Understanding the value of program analysis tools*, OOPSLA practitioner report, 2007

$$\sigma_{end} \quad = \quad \{x \mid x \text{ holds part of the program result}\}$$

We can now define flow functions for live variable analysis. We can do this simply using GEN and KILL sets:

$$\begin{aligned} \text{KILL}_{LV}[\![I]\!] &= \{x \mid I \text{ defines } x\} \\ \text{GEN}_{LV}[\![I]\!] &= \{x \mid I \text{ uses } x\} \end{aligned}$$

We would compute dataflow analysis information for the program shown above as follows. Note that we iterate over the program backwords, i.e. reversing control flow edges between instructions. For each instruction, the corresponding row in our table will hold the information after we have applied the flow function—that is, the variables that are live immediately *before* the statement executes:

stmt	worklist	defs
end	7	$\{z\}$
7	3	$\{z\}$
3	6,2	$\{z, y\}$
6	5,2	$\{z, y\}$
5	4,2	$\{z, y\}$
4	3,2	$\{z, y\}$
3	2	$\{z, y\}$
2	1	$\{y\}$
1	$\varnothing$	$\{x\}$

Lecture Notes:
Program Analysis Correctness

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

Lecture 5

1 Termination

As we think about the correctness of program analysis, let us first think more carefully about the situations under which program analysis will terminate. We will come back later in the lecture to ensuring that the final result of analysis is correct.

In a previous lecture we analyzed the performance of Kildall's worklist algorithm. A critical part of that performance analysis was the the observation that running a flow function always either leaves the dataflow analysis information unchanged, or makes it more approximate—that is, it moves the current dataflow analysis results up in the lattice. The dataflow values at each program point describe an *ascending chain*:

Definition (Ascending Chain). A sequence σ_k is an *ascending chain* iff $n \leqslant m$ implies $\sigma_n \sqsubseteq \sigma_m$

We can can define the height of an ascending chain, and of a lattice, in order to bound the number of new analysis values we can compute at each program point:

Definition (Height of an Ascending Chain). An ascending chain σ_k has finite height h if it contains $h + 1$ distinct elements.

Definition (Height of a Lattice). An lattice $(L, \sqsubseteq)$ has finite height h if there is an ascending chain in the lattice of height h, and no ascending chain in the lattice has height greater than h

We can now show that for a lattice of finite height, the worklist algorithm is guaranteed to terminate. We do so by showing that the dataflow anysis information at each program point follows an ascending chain. Consider the following version of the worklist algorithm:

```
forall (Instruction i ∈ program)
    σ[i] = ⊥
σ[beforeStart] = initialDataflowInformation
worklist = { firstInstruction }

while worklist is not empty
    take an instruction i off the worklist
        var thisInput = ⊥
        forall (Instruction j∈ successors(i))
                thisInput = thisInput ⊔ σ[j]
    let newOutput = flow(i, thisInput)
        if (newOutput ≠ σ[i])
                σ[i] = newOutput
                worklist = worklist ∪ successors(i)
```

We can make the argument for termination inductively. At the beginning of the analysis, the analysis information at every program point (other than the start) is $\perp$. Thus the first time we run each flow function, the result will be at least as high in the lattice as what was there before ($\perp$). We will run the flow function for a given instruction again at a program point only if the dataflow analysis information just before that instruction changes. Assume that the previous time we ran the flow function, we had input information σ_i and output information σ_o. Now we are running it again because the input dataflow analysis information has changed to some new σ_i'—and by the induction hypothesis we assume it is higher in the lattice than before, i.e. $\sigma_i \sqsubseteq \sigma_i'$. What we need to show is that the output information σ_o' is at least as high in the lattice as the old output information σ_o—that is, we must show that $\sigma_o \sqsubseteq \sigma_o'$. This will be true if our flow functions are monotonic:

Definition (Monotonicity). A function f is *monotonic* iff $\sigma_1 \sqsubseteq \sigma_2$ implies $f(\sigma_1) \sqsubseteq f(\sigma_2)$

Now we can state the termination theorem:

Theorem 1 (Dataflow Analysis Termination)**.** *If a dataflow lattice $(L, \sqsubseteq)$ has finite height, and the corresponding flow functions are monotonic, the worklist*

algorithm above will terminate.

Proof. Follows the logic given above when motivating monotonicity. Monotonicity implies that the dataflow value at each program point i will increase each time $\sigma[i]$ is assigned. This can happen a maximum of h times for each program point, where h is the height of the lattice. This bounds the number of elements added to the worklist to $h * e$ where e is the number of edges in the program's control flow graph. Since we remove one element of the worklist for each time through the loop, we will execute the loop at most $h * e$ times before the worklist is empty. Thus the algorithm will terminate. □

2 An Abstract Machine for WHILE3ADDR

In order to reason about the correctness of a program analysis, we need a clear definition of what a program means. There are many ways of giving such definitions; the most common technique in industry is to define a language using an English document, such as the Java Language Specification. However, natural language specifications, while accessible to all programmers, are often imprecise. This imprecision can lead to many problems, such as incorrect or incompatible compiler implementions, but more importantly for our purposes, analyses that give incorrect results.

A better alternative, from the point of view of reasoning precisely about programs, is a formal definition of program semantics. In this class we will deal with *operational semantics*, so named because they show how programs operate. In particular, we will use a form of operational semantics known as an *abstract machine*, in which the semantics mimics, at a high level, the operation of the computer that is executing the program, including a program counter, values for program variables, and (eventually) a representation of the heap. Such a semantics also reflects the way that techniques such as dataflow analysis or Hoare Logic reason about the program, so it is convenient for our purposes.

We now define an abstract machine that evaluates programs in WHILE3ADDR. A configuration c of the abstract machine includes the stored program P (which we will generally treat implicitly), along with an environment E that defines a mapping from variables to values (which for now are just numbers) and the current program counter n representing the next instruction to be executed:

$$
\begin{aligned}
E &\in \mathit{Var} \to \mathbb{Z} \\
c &\in E \times \mathbb{N}
\end{aligned}
$$

The abstract machine executes one step at a time, executing the instruction that the program counter points to, and updating the program counter and environment according to the semantics of that instruction. We will represent execution of the abstract machine with a mathematical judgment of the form $P \vdash E, n \rightsquigarrow E', n'$ The judgment reads as follows: "When executing the program P, executing instruction n in the environment E steps to a new environment E' and program counter n'."

We can now define how the abstract machine executes with a series of inference rules. As shown below, an inference rule is made up of a set of judgments above the line, known as premises, and a judgment below the line, known as the conclusion. The meaning of an inference rule is that the conclusion holds if all of the premises hold.

$$
\frac{\mathit{premise}_1 \quad \mathit{premise}_2 \quad \ldots \quad \mathit{premise}_n}{\mathit{conclusion}}
$$

We now consider a simple rule defining the semantics of the abstract machine for WHILE3ADDR in the case of the constant assignment instruction:

$$
\frac{P[n] = x := m}{P \vdash E, n \rightsquigarrow E[x \mapsto m], n + 1} \textit{ step-const}
$$

This rule states that in the case where the nth instruction of the program P (which we look up using $P[n]$) is a constant assignment $x := m$, the abstract machine takes a step to a state in which the environment E is updated to map x to the constant m, written as $E[x \mapsto m]$, and the program counter now points to the instruction at the following address $n + 1$.

We similarly define the remaining rules:

$$\frac{P[n] = x := y}{P \vdash E, n \rightsquigarrow E[x \mapsto E[y]], n+1} \textit{ step-copy}$$

$$\frac{P[n] = x := y \; op \; z \quad E[y] \; \mathbf{op} \; E[z] = m}{P \vdash E, n \rightsquigarrow E[x \mapsto m], n+1} \textit{ step-arith}$$

$$\frac{P[n] = \text{goto } m}{P \vdash E, n \rightsquigarrow E, m} \textit{ step-goto}$$

$$\frac{P[n] = \text{if } x \; op_r \; 0 \text{ goto } m \quad E[x] \; \mathbf{op_r} \; 0 = true}{P \vdash E, n \rightsquigarrow E, m} \textit{ step-iftrue}$$

$$\frac{P[n] = \text{if } x \; op_r \; 0 \text{ goto } m \quad E[x] \; \mathbf{op_r} \; 0 = false}{P \vdash E, n \rightsquigarrow E, n+1} \textit{ step-iffalse}$$

3 Correctness

Now that we have a model of program execution for WHILE3ADDR we can think more precisely about what it means for an analysis of a WHILE3ADDR program to be correct. Intuitively, we would like the program analysis results to correctly describe every actual execution of the program.

We formalize a program execution as a trace:

Definition (Program Trace). A trace T of a program P is a potentially infinite sequence $\{c_0, c_1, ...\}$ of program configurations, where $c_0 = E_0, 1$ is called the initial configuration, and for every $i \geqslant 0$ we have $P \vdash c_i \rightsquigarrow c_{i+1}$.

Given this definition, we can formally define soundness:

Definition (Dataflow Analysis Soundness). The result $\{\sigma_i \mid i \in P\}$ of a program analysis running on program P is sound iff, for all traces T of P, for all i such that $0 \leqslant i < \textit{length}(T)$, $\alpha(c_i) \sqsubseteq \sigma_{n_i}$

In this definition, just as c_i is the program configuration immediately before executing instruction n_i as the ith program step, σ_i is the dataflow analysis information immediately before instruction n_i.

Exercise 1. Consider the following (incorrect) flow function for zero analysis:

$$f_Z[\![x := y + z]\!](\sigma) = [x \mapsto Z]\sigma$$

Give an example of a program and a concrete trace that illustrates that this flow function is unsound.

The key to designing a sound analysis is making sure that the flow functions map abstract information before each instruction to abstract information after that instruction in a way that matches the instruction's concrete semantics. Another way of saying this is that the manipulation of the abstract state done by the analysis should reflect the manipulation of the concrete machine state done by the executing instruction. We can formalize this as a *local soundness* property.

Definition (Local Soundness). A flow function f is *locally sound* iff $P \vdash c_i \rightsquigarrow c_{i+1}$ and $\alpha(c_i) \sqsubseteq \sigma_i$ and $f[\![P[n_i]]\!](\sigma_i) = \sigma_{i+1}$ implies $\alpha(c_{i+1}) \sqsubseteq \sigma_{i+1}$

Intuitively, if we take any concrete execution of a program instruction, map the input machine state to the abstract domain using the abstraction function, find that the abstracted input state is described by the analysis input information, and apply the flow function, we should get a result that correctly accounts for what happens if we map the actual output machine state to the abstract domain.

Exercise 2. Consider again the incorrect zero analysis flow function described above. Specify an input state c_i and show use that input state that the flow function is not locally sound.

We can now show prove that the flow functions for zero analysis are locally sound. Although technically the abstraction function α accepts a complete program configuration (E, n), for zero analysis we ignore the n component and so in the proof below we will simply focus on the environment E. We show the cases for a couple of interesting syntax forms; the rest are either trivial or analogous:

Case $f_Z[\![x := 0]\!](\sigma_i) = [x \mapsto Z]\sigma_i$:
Assume $c_i = E, n$ and $\alpha(E) = \sigma_i$
Thus $\sigma_{i+1} = f_Z[\![x := 0]\!](\sigma_i) = [x \mapsto Z]\alpha(E)$
$c_{i+1} = [x \mapsto 0]E, n + 1$ by rule *step-const*

Now $\alpha([x \mapsto 0]E) = [x \mapsto Z]\alpha(E)$ by the definition of α.
Therefore $\alpha(c_{i+1}) = \sigma_{i+1}$ which finishes the case.

Case $f_Z[\![x := m]\!](\sigma_i) = [x \mapsto N]\sigma_i \quad$ where $m \neq 0$:
Assume $c_i = E, n$ and $\alpha(E) = \sigma_i$
Thus $\sigma_{i+1} = f_Z[\![x := m]\!](\sigma_i) = [x \mapsto N]\alpha(E)$
$c_{i+1} = [x \mapsto m]E, n + 1$ by rule *step-const*
Now $\alpha([x \mapsto m]E) = [x \mapsto N]\alpha(E)$ by the definition of α and the assumption that $m \neq 0$.
Therefore $\alpha(c_{i+1}) = \sigma_{i+1}$ which finishes the case.

Case $f_Z[\![x := y \; op \; z]\!](\sigma_i) = [x \mapsto ?]\sigma_i$:
Assume $c_i = E, n$ and $\alpha(E) = \sigma_i$
Thus $\sigma_{i+1} = f_Z[\![x := y \; op \; z]\!](\sigma_i) = [x \mapsto ?]\alpha(E)$
$c_{i+1} = [x \mapsto k]E, n + 1$ for some k by rule *step-const*
Now $\alpha([x \mapsto k]E) \sqsubseteq [x \mapsto ?]\alpha(E)$ because the map is equal for all keys except x, and for x we have $\alpha_{simple}(k) \sqsubseteq_{simple} ?$ for all k, where α_{simple} and $\sqsubseteq_{simple}$ are the unlifted versions of α and $\sqsubseteq$, i.e. they operate on individual values rather than maps.
Therefore $\alpha(c_{i+1}) \sqsubseteq \sigma_{i+1}$ which finishes the case.

Exercise 3. Prove the case for $f_Z[\![x := y]\!](\sigma) = [x \mapsto \sigma(y)]\sigma$.

We can also show that zero analysis is monotone. Again, we give some of the more interesting cases:

Case $f_Z[\![x := 0]\!](\sigma) = [x \mapsto Z]\sigma$:
Assume we have $\sigma_1 \sqsubseteq \sigma_2$
Since $\sqsubseteq$ is defined pointwise, we know that $[x \mapsto Z]\sigma_1 \sqsubseteq [x \mapsto Z]\sigma_2$

Case $f_Z[\![x := y]\!](\sigma) = [x \mapsto \sigma(y)]\sigma$:
Assume we have $\sigma_1 \sqsubseteq \sigma_2$
Since $\sqsubseteq$ is defined pointwise, we know that $\sigma_1(y) \sqsubseteq_{simple} \sigma_2(y)$
Therefore, using the pointwise definition of $\sqsubseteq$ again, we also obtain $[x \mapsto \sigma_1(y)]\sigma_1 \sqsubseteq [x \mapsto \sigma_2(y)]\sigma_2$

Now we can show that local soundness can be used to prove the global soundness of a dataflow analysis. To do so, let us formally define the state of the dataflow analysis at a fixed point:

Definition (Fixed Point). A dataflow analysis result $\{\sigma_i \mid i \in P\}$ is a fixed point iff $\sigma_0 \sqsubseteq \sigma_1$ where σ_0 is the initial analysis information and σ_1 is the dataflow result before the first instruction, and for each instruction i we have $\sigma_i = \bigsqcup_{j \in predecessors(i)} f[\![P[j]]\!](\sigma_j)$.

And now the main result we will use to prove program analyses correct:

Theorem 2 (Local Soundness implies Global Soundness)**.** *If a dataflow analysis's flow function f is monotonic and locally sound, and for all traces T we have $\alpha(c_0) \sqsubseteq \sigma_0$ where σ_0 is the initial analysis information, then any fixed point $\{\sigma_i \mid i \in P\}$ of the analysis is sound.*

Proof. Consider an arbitrary program trace T. The proof is by induction on the program configurations $\{c_i\}$ in the trace.

Case c_0:
$\alpha(c_0) \sqsubseteq \sigma_0$ by assumption.
$\sigma_0 \sqsubseteq \sigma_{n_0}$ by the definition of a fixed point.
$\alpha(c_0) \sqsubseteq \sigma_{n_0}$ by the transitivity of $\sqsubseteq$.

Case c_{i+1}:
$\alpha(c_i) \sqsubseteq \sigma_{n_i}$ by the induction hypothesis.
$P \vdash c_i \rightsquigarrow c_{i+1}$ by the definition of a trace.
$\alpha(c_{i+1}) \sqsubseteq f[\![P[n_i]]\!](\alpha(c_i))$ by local soundness.
$f[\![P[n_i]]\!](\alpha(c_i)) \sqsubseteq f[\![P[n_i]]\!](\sigma_{n_i})$ by monotonicity of f.
$\sigma_{n_{i+1}} = f[\![P[n_i]]\!](\sigma_{n_i}) \sqcup ...$ by the definition of fixed point.
$f[\![P[n_i]]\!](\sigma_{n_i}) \sqsubseteq \sigma_{n_{i+1}}$ by the properties of $\sqcup$.
$\alpha(c_{i+1}) \sqsubseteq \sigma_{n_{i+1}}$ by the transitivity of $\sqsubseteq$.

□

Since we previously proved that Zero Analysis is locally sound and that its flow functions are monotonic, we can use this theorem to conclude that the analysis is sound. This means, for example, that Zero Analysis will never neglect to warn us if we are dividing by a variable that could be zero.

Lecture Notes:
Widening Operators and Collecting Semantics

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

Lecture 7

1 Interval Analysis

Let us consider a program analysis that might be suitable for array bounds checking, namely *interval analysis*. As the name suggests, interval analysis tracks the interval of values that each variable might hold. We can define a lattice, initial dataflow information, and abstraction function as follows:

$$\begin{array}{rcl} L & = & \mathbb{Z}_\infty \times \mathbb{Z}_\infty \qquad \text{where } \mathbb{Z}_\infty = \mathbb{Z} \cup \{-\infty, \infty\} \\ [l_1, h_1] \sqsubseteq [l_2, h_2] & \textit{iff} & l_2 \leqslant_\infty l_1 \wedge h_1 \leqslant_\infty h_2 \\ [l_1, h_1] \sqcup [l_2, h_2] & = & [min_\infty(l_1, l_2), max_\infty(h_1, h_2)] \\ \top & = & [-\infty, \infty] \\ \bot & = & [\infty, -\infty] \\ \sigma_0 & = & \top \\ \alpha(x) & = & [x, x] \end{array}$$

In the above, we have extended the $\leqslant$ operator and the *min* and *max* functions to handle sentinels representing positive and negative infinity in the obvious way. For example $-\infty \leqslant_\infty n$ for all $n \in \mathbb{Z}$. For convenience we write the empty interval $\bot$ as $[\infty, -\infty]$.

The lattice above is defined to capture the range of a single variable. As usual, we can lift this lattice to a map from variables to interval lattice elements. Thus we have dataflow information $\sigma \in \mathbf{Var} \to L$

We can also define a set of flow functions. Here we provide one for addition; the rest should be easy for the reader to develop:

$$f_I[\![x := y + z]\!](\sigma) = [x \mapsto [l, h]]\sigma \quad \text{where } l = \sigma(y).low +_\infty \sigma(z).low \text{ and } h = \sigma(y).high +_\infty \sigma(z).high$$

$$f_I[\![x := y + z]\!](\sigma) = \sigma \quad \text{where } \sigma(y) = \bot \vee \sigma(z) = \bot$$

In the above we have extended mathematical $+$ to operate over the sentinels for $\infty, -\infty$, for example such that $\forall x \neq -\infty : \infty + x = \infty$. We define the second case of the flow function to handle the case where one argument is $\bot$, possibly resulting in the undefined case $-\infty + \infty$.

If we run this analysis on a program, whenever we come to an array dereference, we can check whether the interval produced by the analysis for the array index variable is within the bounds of the array. If not, we can issue a warning about a potential array bounds violation.

Just one practical problem remains. Consider, what is the height of the lattice defined above, and what consequences does this have for our analysis in practice?

2 The Widening Operator

As in the example of interval analysis, there are times in which it is useful to define a lattice of infinite height. We would like to nevertheless find a mechanism for ensuring that the analysis will terminate. One way to do this is to find situations where the lattice may be ascending an infinite chain at a given program point, and effectively shorten the chain to a finite height. We can do so with a *widening operator*.

In order to understand the widening operator, consider applying interval analysis to the program below:

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

If we use the worklist algorithm, solving strongly connected components first, the analysis will run as follows:

stmt	worklist	x	y
0	1	$\top$	$\top$
1	2	[0,0]	$\top$
2	3,5	[0,0]	$\top$
3	4,5	[1,1]	$\top$
4	2,5	[1,1]	$\top$
2	3,5	[0,1]	$\top$
3	4,5	[1,2]	$\top$
4	2,5	[1,2]	$\top$
2	3,5	[0,2]	$\top$
3	4,5	[1,3]	$\top$
4	2,5	[1,3]	$\top$
2	3,5	[0,3]	$\top$
...			

Let us consider the sequence of interval lattice elements for x immediately after statement 2. Counting the original lattice value as $\perp$ (not shown explicitly in the trace above), we can see it is the ascending chain $\perp, [0,0], [0,1], [0,2], [0,3], ...$ Recall that ascending chain means that each element of the sequence is higher in the lattice than the previous element. In the case of interval analysis, [0,2] (for example) is higher than [0,1] in the lattice because the latter interval is contained within the former. Given mathematical integers, this chain is clearly infinite; therefore our analysis is not guaranteed to terminate (and indeed it will not in practice).

A widening operator's purpose is to compress such infinite chains to finite length. The widening operator considers the most recent two elements in a chain. If the second is higher than the first, the widening operator can choose to jump up in the lattice, potentially skipping elements in the chain. For example, one way to cut the ascending chain above down to a finite height is to observe that the upper limit for x is increasing, and therefore assume the maximum possible value ∞ for x. Thus we will have the new chain $\perp, [0,0], [0,\infty], [0,\infty], ...$ which has already converged after the third element in the sequence.

The widening operator gets its name because it is an upper bound operator, and in many lattices, higher elements represent a wider range of program values.

We can define the example widening operator given above more formally as follows:

$$
\begin{array}{lll}
W(\bot, l_{current}) & = l_{current} & \\
W([l_1, h_1], [l_2, h_2]) & = [min_W(l_1, l_2), max_W(h_1, h_2)] & \\
 & \text{where } min_W(l_1, l_2) = l_1 & \text{if } l_1 \leqslant l_2 \\
 & \text{and } min_W(l_1, l_2) = -\infty & \text{otherwise} \\
 & \text{where } max_W(h_1, h_2) = h_1 & \text{if } h_1 \geqslant h_2 \\
 & \text{and } max_W(h_1, h_2) = \infty & \text{otherwise}
\end{array}
$$

Applying this widening operator each time just before analysing instruction 2 gives us the following sequence:

stmt	worklist	x	y
0	1	$\top$	$\top$
1	2	[0,0]	$\top$
2	3,5	[0,0]	$\top$
3	4,5	[1,1]	$\top$
4	2,5	[1,1]	$\top$
2	3,5	[0,∞]	$\top$
3	4,5	[1,∞]	$\top$
4	2,5	[1,∞]	$\top$
2	5	[0,∞]	$\top$
5	$\varnothing$	[0,∞]	[0,0]

Before we analyze instruction 2 the first time, we compute $W(\bot, [0, 0]) = [0, 0]$ using the first case of the definition of W. Before we analyze instruction 2 the second time, we compute $W([0, 0], [0, 1]) = [0, \infty]$. In particular, the lower bound 0 has not changed, but since the upper bound has increased from $h_1 = 0$ to $h_2 = 1$, the max_W helper function sets the maximum to ∞. After we go through the loop a second time we observe that iteration has converged at a fixed point. We therefore analyze statement 5 and we are done.

Let us consider the properties of widening operators more generally. A widening operator $W(l_{previous}{:}L,\ l_{current}{:}L) : L$ accepts two lattice elements, the previous lattice value $l_{previous}$ at a program location and the current lattice value $l_{current}$ at the same program location. It returns a new lattice value that will be used in place of the current lattice value.

We require two properties of widening operators. The first is that the widening operator must return an upper bound of its operands. Intuitively, this is required for monotonicity: if the operator is applied to an

ascending chain, an ascending chian should be a result. Formally, we have $\forall l_{previous}, l_{current} : l_{previous} \sqsubseteq W(l_{previous}, l_{current}) \wedge l_{current} \sqsubseteq W(l_{previous}, l_{current})$.

The second property is that when the widening operator is applied to an ascending chain l_i, the resulting ascending chain l_i^W must be of finite height. Formally we define $l_0^W = l_0$ and $\forall i > 0 : l_i^W = W(l_{i-1}^W, l_i)$. This property ensures that when we apply the widening operator, it will ensure that the analysis terminates.

Where can we apply the widening operator? Clearly it is safe to apply anywhere, since it must be an upper bound and therefore can only raise the analysis result in the lattice, thus making the analysis result more conservative. However, widening inherently causes a loss of precision. Therefore it is better to apply it only when necessary. One solution is to apply the widening operator only at the heads of loops, as in the example above. Loop heads (or their equivalent, in unstructured control flow) can be inferred even from low-level three address code—see a compiler text such as Appel and Palsberg's *Modern Compiler Implementation in Java*.

We can use a somewhat smarter version of this widening operator with the insight that the bounds of a lattice are often related to constants in the program. Thus if we have an ascending chain $\bot, [0,0], [0,1], [0,2], [0,3], ...$ and the constant 10 is in the program, we might change the chain to $\bot, [0,0], [0,10], ...$ If we are lucky, the chain will stop ascending that that point: $\bot, [0,0], [0,10], [0,10], ...$ If we are not so fortunate, the chain will continue and eventually stabilize at $[0,\infty]$ as before: $\bot, [0,0], [0,10], [0,\infty]$.

If the program has the set of constants K, we can define a widening operator as follows:

$$
\begin{array}{lll}
W(\bot, l_{current}) & = l_{current} & \\
W([l_1,h_1],[l_2,h_2]) & = [min_K(l_1,l_2), max_K(h_1,h_2)] & \\
 & \text{where } min_K(l_1,l_2) = l_1 & \text{if } l_1 \leqslant l_2 \\
 & \text{and } min_K(l_1,l_2) = max(\{k \in K | k \leqslant l_2\}) & \text{otherwise} \\
 & \text{where } max_K(h_1,h_2) = h_1 & \text{if } h_1 \geqslant h_2 \\
 & \text{and } max_K(h_1,h_2) = min(\{k \in K | k \geqslant h_2\} & \text{otherwise}
\end{array}
$$

We can now analyze a program with a couple of constants and see how this approach works:

$1:$ $x := 0$
$2:$ $y := 1$
$3:$ if $x = 10$ goto 7
$4:$ $x := x + 1$
$5:$ $y := y - 1$
$6:$ goto 3
$7:$ goto 7

Here the constants in the program are $0, 1$ and 10. The analysis results are as follows:

stmt	worklist	x	y
0	1	$\top$	$\top$
1	2	[0,0]	$\top$
2	3	[0,0]	$[1,1]$
3	4,7	$[0,0]_F, \bot_T$	$[1,1]$
4	5,7	[1,1]	$[1,1]$
5	6,7	[1,1]	$[0,0]$
6	3,7	[1,1]	$[0,0]$
3	4,7	$[0,1]_F, \bot_T$	$[0,1]$
4	5,7	[1,2]	$[0,1]$
5	6,7	[1,2]	$[-1,0]$
6	3,7	[1,2]	$[-1,0]$
3	4,7	$[0,9]_F, [10,10]_T$	$[-\infty,1]$
4	5,7	[1,10]	$[-\infty,1]$
5	6,7	[1,10]	$[-\infty,0]$
6	3,7	[1,10]	$[-\infty,0]$
3	7	$[0,9]_F, [10,10]_T$	$[-\infty,1]$
7	$\varnothing$	[10,10]	$[-\infty,1]$

Applying the widening operation the first time we get to statement 3 has no effect, as the previous analysis value was $\bot$. The second time we get to statement 3, the range of both x and y has been extended, but both are still bounded by constants in the program. The third time we get to statement 3, we apply the widening operator to x, whose abstract value has gone from [0,1] to [0,2]. The widened abstract value is [0,10], since 10 is the smallest constant in the program that is at least as large as 2. For y we must widen to $[-\infty, 1]$. After one more iteration the analysis stabilizes.

In this example I have assumed a flow function for the if instruction that propagates different interval information depending on whether the

branch is taken or not. In the table, we list the branch taken information for x as $\bot$ until x reaches the range in which it is feasible to take the branch. $\bot$ can be seen as a natural representation for dataflow values that propagate along a path that is infeasible.

3 A Collecting Semantics for Reaching Definitions

The approach to dataflow analysis correctness outlined in the previous two lectures generalizes naturally when we have a lattice that can be directly abstracted from program configurations c from our execution semantics. Sometimes, however, we cannot get the information we need directly from a particular state in program execution. An example is reaching definitions. When we look at a particular execution of an instruction I, we cannot see where the values of variables used in I were last defined.

In order to solve this problem, we need to augment our semantics with additional information that captures the required information. For example, for the reaching definitions analysis, we need to know, at any point in a particular execution, which definition reaches the current location for each program variable in scope.

We call a version of the program semantics that has been augmented with additional information necessary for some particular analysis a *collecting semantics*. For reaching definitions, we can define a collecting semantics with a version of the environment E, which we will call E_{RD}, that has been extended with a index n indicating the location where each variable was last defined.

$$E_{RD} \quad \in \quad \mathit{Var} \rightarrow \mathbb{Z} \times \mathbb{N}$$

We can now extend the semantics to track this information. We show only the rules that differ from those described in the earlier lectures:

$$\frac{P[n] = x := m}{P \vdash E, n \rightsquigarrow E[x \mapsto m, n], n+1} \textit{ step-const}$$

$$\frac{P[n] = x := y}{P \vdash E, n \rightsquigarrow E[x \mapsto E[y], n], n+1} \textit{ step-copy}$$

$$\frac{P[n] = x := y \textit{ op } z \quad E[y] \textbf{ op } E[z] = m}{P \vdash E, n \rightsquigarrow E[x \mapsto m, n], n+1} \textit{ step-arith}$$

Essentially, each rule that defines a variable records the current location as the latest definition of that variable.

Now we can define an abstraction function for reaching definitions from this collecting semantics:

$$\alpha_{RD}(E_{RD}, n) = \{m \mid \exists x \in \mathit{domain}(E_{RD}) \text{ such that } E_{RD}(x) = i, m\}$$

From this point, reasoning about the correctness of reaching definitions proceeds analogously to the reasoning for zero analysis outlined in the previous lectures.

Formulating a collecting semantics is even trickier for some analyses, but it can be done with a little thought. Consider live variable analysis. The collecting semantics requires us to know, for each execution of the program, which variables currently in scope will be used before they are defined in the remainder of the program. We can compute this semantics by assuming a (possibly infinite) trace for a program run, then specifying the set of live variables at every point in that trace based on the trace going forward from that point. This semantics, specified in terms of traces rather than a set of inference rules, can then be used in the definition of an abstraction function and used to reason about the correctness of live variables analysis.

Lecture Notes: Interprocedural Analysis

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

Lecture 8

1 Interprocedural Analysis

Interprocedural analysis concerns analyzing a program with multiple procedures, ideally taking into account the way that information flows among those procedures.

1.1 Default Assumptions

Our first approach assumes a default lattice value for all arguments L_a and a default value for procedure results L_r

We check the assumption holds when analyzing a call instruction or a return instruction (trivial if $L_a = L_r = \top$)

We use the assumption when analyzing the result of a call instruction or starting the analysis of a method. For example, we have $\sigma_0 = \{x \mapsto L_a \mid x \in \mathbf{Var}\}$.

Here is a sample flow function for call and return instructions:

$$\begin{aligned} f[\![x := g(y)]\!](\sigma) &= [x \mapsto L_r]\sigma && \text{where } \sigma(y) \sqsubseteq L_a \\ f[\![\text{return } x]\!](\sigma) &= \sigma && \text{where } \sigma(x) \sqsubseteq L_r \end{aligned}$$

We can apply zero analysis to the following function, using $L_a = L_r = \top$:

$$
\begin{array}{ll}
1: & \text{procedure } \textit{divByX}(x : int) : int \\
2: & \quad y := 10/x \\
3: & \quad \text{return } y \\
4: & \text{procedure } \textit{main}() \\
5: & \quad z := 5 \\
6: & \quad w := \textit{divByX}(z)
\end{array}
$$

We can avoid the error by using a more optimistic assumption $L_a = L_r = NZ$. But then we get a problem with the following program:

$$
\begin{array}{ll}
1: & \text{procedure } \textit{double}(x : int) : int \\
2: & \quad y := 2 * x \\
3: & \quad \text{return } y \\
4: & \text{procedure } \textit{main}() \\
5: & \quad z := 0 \\
6: & \quad w := \textit{double}(z)
\end{array}
$$

1.2 Local vs. global variables

The above analysis assumes we have only local variables. If we have global variables, we must make conservative assumptions about them too. Assume globals should always be described by some lattice value L_g at procedure boundaries. We can extend the flow functions as follows:

$$
\begin{array}{ll}
f[\![x := g(y)]\!](\sigma) & = [x \mapsto L_r][z \mapsto L_g \mid z \in \textbf{Globals}]\sigma \\
& \quad \text{where } \sigma(y) \sqsubseteq L_a \wedge \forall z \in \textbf{Globals} : \sigma(z) \sqsubseteq L_g \\
f[\![\text{return } x]\!](\sigma) & = \sigma \\
& \quad \text{where } \sigma(x) \sqsubseteq L_r \wedge \forall z \in \textbf{Globals} : \sigma(z) \sqsubseteq L_g
\end{array}
$$

1.3 Annotations

An alternative approach is using annotations. This allows us to choose different argument and result assumptions for different procedures. Flow functions might look like:

$$\begin{array}{lll} f[\![x := g(y)]\!](\sigma) & = [x \mapsto \mathit{annot}[\![g]\!].r]\sigma & \text{where } \sigma(y) \sqsubseteq \mathit{annot}[\![g]\!].a \\ f[\![\text{return } x]\!](\sigma) & = \sigma & \text{where } \sigma(x) \sqsubseteq \mathit{annot}[\![g]\!].r \end{array}$$

Now we can verify that both of the above programs are safe. But some programs remain difficult:

```
1 : procedure double(x : int @⊤) : int @⊤
2 :     y := 2 * x
3 :     return y
4 : procedure main()
5 :     z := 5
6 :     w := double(z)
7 :     z := 10/w
```

Annotations can be extended in a natural way to handle global variables.

1.4 Interprocedural Control Flow Graph

An approach that avoids the burden of annotations, and can capture what a procedure actually does as used in a particular program, is building a control flow graph for the entire program, rather than just one procedure. To make this work, we handle call and return instructions specially as follows:

- We add additional edges to the control flow graph. For every call to function g, we add an edge from the call site to the first instruction of g, and from every return statement of g to the instruction following that call.
- When analyzing the first statement of a procedure, we generally gather analysis information from each predecessor as usual. However, we take out all dataflow information related to local variables in the callers. Furthermore, we add dataflow information for parameters in the callee, initializing their dataflow values according to the actual arguments passed in at each call site.

- When analyzing an instruction immediately after a call, we get dataflow information about local variables from the previous statement. Information about global variables is taken from the return sites of the function that was called. Information about the variable that the result of the function call was assigned to comes from the dataflow information about the returned value.

Now the example described above can be successfully analyzed. However, other programs still cause problems:

$1:$ procedure *double*$(x : int\ @\top) : int\ @\top$
$2:$ $\quad y := 2 * x$
$3:$ $\quad$ return y
$4:$ procedure *main*$()$
$5:$ $\quad z := 5$
$6:$ $\quad w :=$ *double*(z)
$7:$ $\quad z := 10/w$
$8:$ $\quad z := 0$
$9:$ $\quad w :=$ *double*(z)

1.5 Context Sensitive Analysis

Context-sensitive analysis analyzes a function either multiple times, or parametrically, so that the analysis results returned to different call sites reflect the different analysis results passed in at those call sites.

We can get context sensitivity just by duplicating all callees. But this works only for non-recursive programs.

A simple solution is to build a summary of each function, mapping dataflow input information to dataflow output information. We will analyze each function once for each *context*, where a context is an abstraction for a set of calls to that function. At a minimum, each context must track the input dataflow information to the function.

Let's look at how this approach allows the program given above to be proven safe by zero analysis.

[Example given in class]

Things become more challenging in the presence of recursive functions, or more generally mutual recursion. Let us consider context-sensitive interprocedural constant propagation analysis of the factorial function called by main. We are not focused on the intraprocedural part of the analysis so we will just show the function in the form of Java or C source code:

```
int fact(int x) {
    if (x == 1)
        return 1;
    else
        return x * fact(x-1);
}
void main() {
    int y = fact(2);
    int z = fact(3);
    int w = fact(getInputFromUser());
}
```

We can analyze the first two calls to fact using the following algorithm:

```
begin()
    // initial context is main() with argument assumptions
    context = get the initial program context
    analyze(context)

analyze(context)
    newResults = intraprocedural(context)
    resultsMap.put(context, newResults)
    return newResults

// called by intraprocedural analysis of "context"
analyzeCall(context, callInfo) : AnalysisResult
    calleeContext = computeCalleeContext(context, callinfo)
    results = getResultsFor(calleeContext)
    return results

getResultsFor(context)
    results = resultsMap.get(context)
    if (results != bottom)
        return results;
    else
        return analyze(context)

computeCalleeContext(callingContext, callinfo)
    // calling context is just the input information
    return callinfo.inputInfo
```

The resultsMap and the function getResultsFor() acts as a cache for anal-

ysis results, so that when fib(3) invokes fib(2), the results from the prior call fib(2) can be reused.

For the third call to fib, the argument is determined at runtime and so constant propagation uses $\top$ for the calling context. In this case the recursive call to fib() also has $\top$ as the calling context. But we cannot look up the result in the cache yet as analysis of fib() with $\top$ has not completed. Thus the algorithm above will attempt to analyze fib() with $\top$ again, and it will therefore not terminate.

We can solve the problem by applying the same idea as in intraprocedural analysis. The recursive call is a kind of a loop. We can make the initial assumption that the result of the recursive call is $\bot$, which is conceptually equivalent to information coming from the back edge of a loop. When we discover the result is a higher point in the lattice then $\bot$, we reanalyze the calling context (and recursively, all calling contexts that depend on it). The algorithm to do so can be expressed as follows:

```
begin()
    // initial context is main() with argument assumptions
    context = get the initial program context
    analyze(context)
    while context = worklist.remove()
        analyze(context)

analyze(context)
    oldResults = resultsMap.get(context)
    newResults = intraprocedural(context)
    if (newResults != oldResults)
        resultsMap.put(context, newResults)
        for ctx in callingContextsOf(context)
            worklist.add(ctx)
    return newResults

// called by intraprocedural analysis of "context"
analyzeCall(context, callInfo) : AnalysisResult
    calleeContext = computeCalleeContext(context, callinfo)
    results = getResultsFor(calleeContext)
    add context to callingContextsOf(calleeContext)
    return results

getResultsFor(context)
```

```
if (context is currently being analyzed)
    return bottom
results = resultsMap.get(context)
if (results != bottom)
    return results;
else
    return analyze(context)
```

The following example shows that the algorithm generalizes naturally to the case of mutually recursive functions:

```
bar() { if (*) return 2 else return foo() }
foo() { if (*) return 1 else return bar() }

main() { foo(); }
```

The description above considers differentiates calling contexts by the input dataflow information. A historical alternative is to differentiate contexts by their call string: the call site, it's call site, and so forth. In the limit, when considering call strings of arbitrary length, this provides full context sensitivity.

Dataflow analysis results for contexts based on arbitrary-length call strings are as precise as the results for contexts based on separate analysis for each different input dataflow information. The latter strategy can be more efficient, however, because it reuses analysis results when a function is called twice with different call strings but the same input dataflow information.

In practice, both strategies (arbitrary-length call strings vs. input dataflow information) can result in reanalyzing each function so many times that performance becomes unacceptable. Thus multiple contexts must be combined somehow to reduce the number of times each function is analyzed. The call-string approach provides an easy, but naive, way to do this: call strings can be cut off at a certain length. For example, if we have call strings "a b c" and "d e b c" (where c is the most recent call site) with a cutoff of 2, the input dataflow information for these two call strings will be merged and the analysis will be run only once, for the context identified by the common length-two suffix of the strings, "b c". We can illustrate this by redoing the analysis of the fibonacci example. The algorithm is the same as above; however, we use a different implementation of compteCalleeContext that computes the call string suffix and, if it has already been analyzed, merges the incoming dataflow analysis information with what is already there:

```
computeCalleeContext(callingContext, callinfo)
    let oldCallString = callingContext.callString
    let newCallString = suffix(oldCallString ++ callinfo.site,
                               CALL_STRING_CUTOFF)
    let newContext = new Context(newCallString, callinfo.inputInfo)

    // look for a previous analysis with the same call string
    // context identity (and map lookup) is determined
    // by the call string
    if (resultsMap.containsKey(newContext))
        let oldContext = resultsMap.findKey(newContext);
        if (newContext.inputInfo ⋢ oldContext.inputInfo)
                // force reanalysis with joined input information
            resultsMap.removeKey(newContext)
            newContext.inputInfo = newContext.inputInfo
                                 ⊔ oldContext.inputInfo

    return newContext
```

Although this strategy reduces the overall number of analyses, it does so in a relatively blind way. If a function is called many times but we only want to analyze it a few times, we want to group the calls into analyzis contexts so that their input information is similar. Call string context is a hueristic way of doing this that sometimes works well. But it can be wasteful: if two different call strings of a given length happen to have exactly the same input analysis information, we will do an unnecessary extra analysis, whereas it would have been better to spend that extra analysis to differentiate calls with longer call strings that have different analysis information.

Given a limited analysis budget, it is smarter to use hueristics that are directly based on input information. Unfortunately these hueristics are harder to design, but they have the potential to do much better than a call-string based approach. We will look at some examples from the literature to illustrate this later in the course.

Lecture Notes: Pointer Analysis

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

Lecture 9

1 Motivation for Pointer Analysis

In programs with pointers, program analysis can become more challenging. Consider constant-propagation analysis of the following program:

$$\begin{array}{ll} 1: & z := 1 \\ 2: & p := \&z \\ 3: & *p := 2 \\ 4: & \text{print } z \end{array}$$

In order to analyze this program correctly we must be aware that at instruction 3 p points to z. If this information is available we can use it in a flow function as follows:

$$f_{CP}[\![*p := y]\!](\sigma) \quad = [z \mapsto \sigma(y)]\sigma \quad \text{where } \textit{must-point-to}(p, z)$$

When we know exactly what a variable x points to, we say that we have must-point-to information, and we can perform a *strong update* of the target variable z, because we know with confidence that assigning to $*p$ assigns to z. A technicality in the rule is quantifying over all z such that p must point to z. How is this possible? It is not possible in C or Java; however, in a language with pass-by-reference, for example C++, it is possible that two names for the same location are in scope.

Of course, it is also possible that we are uncertain to which of several distinct locations p points. For example:

$$
\begin{array}{ll}
1: & z := 1 \\
2: & \textbf{if } (\textit{cond})\ p := \&y \textbf{ else } p := \&z \\
3: & *p := 2 \\
4: & \textbf{print } z
\end{array}
$$

Now constant propagation analysis must conservatively assume that z could hold either 1 or 2. We can represent this with a flow function that uses may-point-to information:

$$f_{CP}[\![*p := y]\!](\sigma) \quad = [z \mapsto \sigma(z) \sqcup \sigma(y)]\sigma \quad \text{where } \textit{may-point-to}(p, z)$$

2 Andersen's Points-To Analysis

Two common kinds of pointer analysis are alias analysis and points-to analysis. Alias analysis computes a set S holding pairs of variables (p, q), where p and q may (or must) point to the same location. On the other hand, points-to analysis, as described above, computes a relation $\textit{points-to}(p, x)$, where p may (or must) point to the location of the variable x. We will focus our study in this lecture on points-to analysis, and will begin with a simple but useful approach originally proposed by Andersen.

Our initial setting will be C programs. We are interested in analyzing instructions that are relevant to pointers in the program. Ignoring for the moment memory allocation and arrays, we can decompose all pointer operations into four instruction types: taking the address of a variable, copying a pointer from one variable to another, assigning through a pointer, and dereferencing a pointer:

$$
\begin{array}{lll}
I & ::= & \ldots \\
 & | & p := \&x \\
 & | & p := q \\
 & | & *p := q \\
 & | & p := *q
\end{array}
$$

Andersen's points-to analysis is a context-insensitive interprocedural analysis. It is also a *flow-insensitive analysis*, that is an analysis that (unlike dataflow analysis) does not take into consideration the order of program statements. Context- and flow-insensitivity are used to improve the performance of the analysis, as precise pointer analysis can be notoriously expensive in practice.

We will formulate Andersen's analysis by generating set constraints which can later be processed by a set constraint solver using a number of technologies. Constraint generation for each statement works as given in the following set of rules. Because the analysis is flow-insensitive, we do not care what order the instructions in the program come in; we simply generate a set of constraints and solve them.

$$\frac{}{[\![p := \&x]\!] \hookrightarrow l_x \in p} \textit{ address-of}$$

$$\frac{}{[\![p := q]\!] \hookrightarrow p \supseteq q} \textit{ copy}$$

$$\frac{}{[\![*p := q]\!] \hookrightarrow *p \supseteq q} \textit{ assign}$$

$$\frac{}{[\![p := *q]\!] \hookrightarrow p \supseteq *q} \textit{ dereference}$$

The constraints generated are all set constraints. The first rule states that a constant location l_x, representation the address of x, is in the set of location pointed to by p. The second rule states that the set of locations pointed to by p must be a superset of those pointed to by q. The last two rules state the same, but take into account that one or the other pointer is dereferenced.

A number of specialized set constraint solvers exist and constraints in the form above can be translated into the input for these. The dereference operation (the $*$ in $*p \supseteq q$) is not standard in set constraints, but it can be encoded—see Fähndrich's Ph.D. thesis for an example of how to encode Andersen's points-to analysis for the BANE constraint solving engine. We will treat constraint-solving abstractly using the following constraint propagation rules:

$$\frac{p \supseteq q \quad l_x \in q}{l_x \in p} \textit{ copy}$$

$$\frac{*p \supseteq q \quad l_r \in p \quad l_x \in q}{l_x \in r} \textit{ assign}$$

$$\frac{p \supseteq *q \quad l_r \in q \quad l_x \in r}{l_x \in p} \textit{ dereference}$$

We can now apply Andersen's points-to analysis to the program above. Note that in this example if Andersen's algorithm says that the set p points to only one location l_z, we have must-point-to information, whereas if the set p contains more than one location, we have only may-point-to information.

We can also apply Andersen's analysis to programs with dynamic memory allocation, such as:

$$\begin{array}{ll} 1: & q := \textit{malloc}_1() \\ 2: & p := \textit{malloc}_2() \\ 3: & p := q \\ 4: & r := \&p \\ 5: & s := \textit{malloc}_3() \\ 6: & *r := s \\ 7: & t := \&s \\ 8: & u := *t \end{array}$$

In this example, the analysis is run the same way, but we treat the memory cell allocated at each *malloc* or *new* statement as an abstract location labeled by the location n of the allocation point. We can use the rules:

$$\frac{}{[\![p := \textit{malloc}_n()]\!] \hookrightarrow l_n \in p} \textit{ malloc}$$

We must be careful because a *malloc* statement can be executed more than once, and each time it executes, a new memory cell is allocated. Unless we have some other means of proving that the malloc executes only once, we must assume that if some variable p only points to one abstract malloc'd location l_n, that is still may-alias information (i.e. p points to only one of the many actual cells allocated at the given program location) and not must-alias information.

Analyzing the efficiency of Andersen's algorithm, we can see that all constraints can be generated in a linear $O(n)$ pass over the program. The solution size is $O(n^2)$ because each of the $O(n)$ variables defined in the program could potentially point to $O(n)$ other variables.

We can derive the execution time from a theorem by David McAllester published in SAS'99. There are $O(n)$ flow constraints generated of the form $p \supseteq q$, $*p \supseteq q$, or $p \supseteq *q$. How many times could a constraint propagation rule fire for each flow constraint? For a $p \supseteq q$ constraint, the rule may fire at most $O(n)$ times, because there are at most $O(n)$ premises of the proper form $l_x \in p$. However, a constraint of the form $p \supseteq *q$ could cause $O(n^2)$ rule firings, because there are $O(n)$ premises each of the form $l_x \in p$ and $l_r \in q$. With $O(n)$ constraints of the form $p \supseteq *q$ and $O(n^2)$ firings for each, we have $O(n^3)$ constraint firings overall. A similar analysis applies for $*p \supseteq q$ constraints. McAllester's theorem states that the analysis with $O(n^3)$ rule firings can be implemented in $O(n^3)$ time. Thus we have derived that Andersen's algorithm is cubic in the size of the program, in the worst case.

2.1 Field-Sensitive Analysis

The algorithm above works in C-like languages for pointers to single memory cells. However, what about when we have a pointer to a struct in C, or an object in an object-oriented language? In this case, we would like the pointer analysis to tell us what each field in the struct or object points to.

A simple solution is to be *field-insensitive*, treating all fields in a struct as equivalent. Thus if p points to a struct with two fields f and g, and we assign:

$$\begin{aligned} 1: &\quad p.f := \&x \\ 2: &\quad p.g := \&y \end{aligned}$$

A field-insensitive analysis would tell us (imprecisely) that $p.f$ could point to y.

In order to be more precise, we can track the contents each field of each abstract location separately. In the discussion below, we assume a setting in which we cannot take the address of a field; this assumption is true for Java but not for C. We can define a new kind of constraints for fields:

$$\frac{}{[\![p := q.f]\!] \hookrightarrow p \supseteq q.f} \textit{ field-read}$$

$$\frac{}{[\![p.f := q]\!] \hookrightarrow p.f \supseteq q} \textit{ field-assign}$$

Now assume that objects (e.g. in Java) are represented by abstract locations l. We can process field constraints with the following rules:

$$\frac{p \supseteq q.f \quad l_q \in q \quad l_f \in l_q.f}{l_f \in p} \textit{ field-read}$$

$$\frac{p.f \supseteq q \quad l_p \in p \quad l_q \in q}{l_q \in l_p.f} \textit{ field-assign}$$

If we run this analysis on the code above, we find that it can distinguish that $p.f$ points to x and $p.g$ points to y.

3 Steensgaard's Points-To Analysis

For large programs, a cubic algorithm is too inefficient. Steensgaard proposed an pointer analysis algorithm that operates in near-linear time, supporting essentially unlimited scalability in practice.

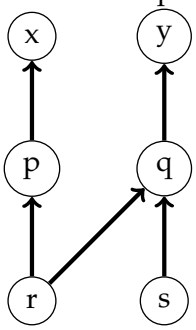
The first challenge in designing a near-linear time points-to analysis is finding a way to represent the results in linear space. This is nontrivial because over the course of program execution, any given pointer p could potentially point to the location of any other variable or pointer q. Representing all of these pointers explicitly will inherently take $O(n^2)$ space.

The solution Steensgaard found is based on using constant space for each variable in the program. His analysis associates each variable p with an abstract location named after the variable. Then, it tracks a single points-to relation between that abstract location p and another one q, to which it may point. Now, it is possible that in some real program p may point to both q and some other variable r. In this situation, Steensgaard's algorithm *unifies* the abstract locations for q and r, creating a single abstract location representing both of them. Now we can track the fact that p may point to either variable using a single points-to relationship.

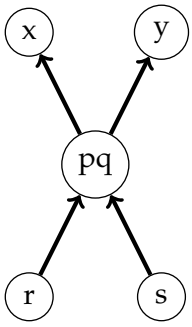
For example, consider the program below:

$$
\begin{array}{ll}
1: & p := \&x \\
2: & r := \&p \\
3: & q := \&y \\
4: & s := \&q \\
5: & r := s
\end{array}
$$

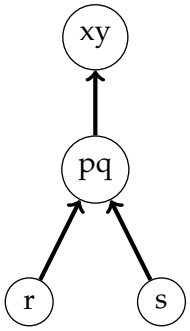
Andersen's points-to analysis would produce the following graph:



But in Steensgaard's setting, when we discover that r could point both to q and to p, we must merge q and p into a single node:



Notice that we have lost precision: by merging the nodes for p and q our graph now implies that s could point to p, which is not the case in the actual program. But we are not done. Now pq has two outgoing arrows, so we must merge nodes x and y. The final graph produced by Steensgaard's algorithm is therefore:



Now let us define Steensgaard's analysis more precisely. We will study a simplified version of the analysis that does not consider function pointers. The analysis can be specified as follows:

$$\frac{}{[\![p := q]\!] \hookrightarrow \mathit{join}(*p, *q)} \textit{ copy}$$

$$\frac{}{[\![p := \&x]\!] \hookrightarrow \mathit{join}(*p, x)} \textit{ address-of}$$

$$\frac{}{[\![p := *q]\!] \hookrightarrow \mathit{join}(*p, **q)} \textit{ dereference}$$

$$\frac{}{[\![*p := q]\!] \hookrightarrow \mathit{join}(**p, *q)} \textit{ assign}$$

With each abstract location p, we associate the abstract location that p points to, denoted $*p$. Abstract locations are implemented as a union-find[1] data structure so that we can merge two abstract locations efficiently. In the rules above, we implicitly invoke *find* on an abstract location before calling *join* on it, or before looking up the location it points to.

The *join* operation essentially implements a union operation on the abstract locations. However, since we are tracking what each abstract location points to, we must update this information also. The algorithm to do so is as follows:

```
join(e1, e2)
    if (e1 == e2)
        return
    e1next = *e1
    e2next = *e2
```

[1] See any algorithms textbook

```
unify(e1, e2)
join(e1next, e2next)
```

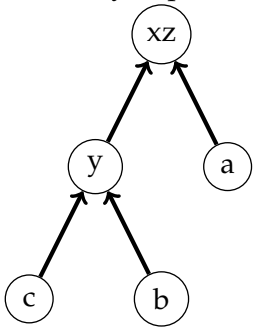
Once again, we implicitly invoke *find* on an abstract location before comparing it for equality, looking up the abstract location it points to, or calling *join* recursively.

As an optimization, Steensgaard does not perform the join if the right hand side is not a pointer. For example, if we have an assignment $[\![p := q]\!]$ and q has not been assigned any pointer value so far in the analysis, we ignore the assignment. If later we find that q may hold a pointer, we must revisit the assignment to get a sound result.

Steensgaard illustrated his algorithm using the following program:

$$
\begin{array}{ll}
1: & a := \&x \\
2: & b := \&y \\
3: & \text{if } p \text{ then} \\
4: & \quad y := \&z \\
5: & \text{else} \\
6: & \quad y := \&x \\
7: & c := \&y
\end{array}
$$

His analysis produces the following graph for this program:



Rayside illustrates a situation in which Andersen must do more work than Steensgaard:

$$
\begin{array}{ll}
1: & q := \&x \\
2: & q := \&y \\
3: & p := q \\
4: & q := \&z
\end{array}
$$

After processing the first three statements, Steensgaard's algorithm will have unified variables x and y, with p and q both pointing to the unified node. In contrast, Andersen's algorithm will have both p and q pointing

to both x and y. When the fourth statement is processed, Steensgaard's algorithm does only a constant amount of work, merging z in with the already-merged xy node. On the other hand, Andersen's algorithm must not just create a points-to relation from q to z, but must also propagate that relationship to p. It is this additional propagation step that results in the significant performance difference between these algorithms.

Analyzing Steensgaard's pointer analysis for efficiency, we observe that each of n statements in the program is processed once. The processing is linear, except for *find* operations on the union-find data structure (which may take amortized time $O(\alpha(n))$ each) and the *join* operations. We note that in the *join* algorithm, the short-circuit test will fail at most $O(n)$ times—at most once for each variable in the program. Each time the short-circuit fails, two abstract locations are unified, at cost $O(\alpha(n))$. The unification assures the short-circuit will not fail again for one of these two variables. Because we have at most $O(n)$ operations and the amortized cost of each operation is at most $O(\alpha(n))$, the overall running time of the algorithm is near linear: $O(n * \alpha(n))$. Space consumption is linear, as no space is used beyond that used to represent abstract locations for all the variables in the program text.

Based on this asymptotic efficiency, Steensgaard's algorithm was run on a 1 million line program (Microsoft Word) in 1996; this was an order of magnitude greater scalability than other pointer analyses known at the time.

Steensgaard's pointer analysis is field-insensitive; making it field-sensitive would mean that it is no longer linear.

4 Adding Context Sensitivity to Andersen's Algorithm

We can define a version of Andersen's points-to algorithm that is context-sensitive. In the following approach, we analyze each function separately for each calling point. The analysis keeps track of the current context, the calling point n of the current procedure. In the constraints, we track separate values for each variable x_n according to the calling context n of the procedure defining it, and we track separate values for each memory location l_n^k according to the calling context n active when that location was allocated at new instruction k. The rules are as follows:

$$\frac{n \vdash p := \mathbf{new}_k\ A}{l_n^k \in p_n}\ \textit{new}$$

$$\frac{n \vdash p := q \quad l_n \in q_n}{l_n \in p_n}\ \textit{copy}$$

$$\frac{n \vdash x.f := y \quad l_x \in x_n \quad l_y \in y_n}{l_y \in l_x.f}\ \textit{field-read}$$

$$\frac{n \vdash x := y.f \quad l_y \in y_n \quad l_z \in l_y.f}{l_z \in x_n}\ \textit{field-assign}$$

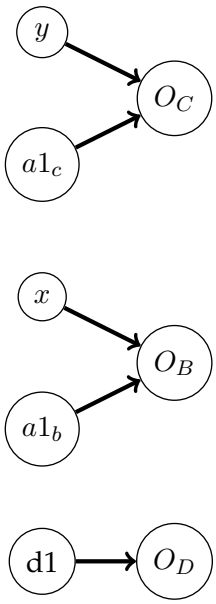
$$\frac{n \vdash f_k(y) \quad l_y \in y_n \quad [\![f(z) = e]\!] \in \textit{Program}}{l_y \in z_k \quad k \vdash e}\ \textit{call}$$

To illustrate this analysis, imagine we have the following code:

```
interface A { void g(); }
class B implements A { void g() { ... } }
class C implements A { void g() { ... } }
class D {
    A f(A a1) { return a1; }
}

// in main()
D d1 = new D();
if (...) {
    A x = d1.f(new B());
        x.g() // which g is called?
else
    A y = d1.f(new C());
        y.g() // which g is called?
```

The analysis produces the following aliasing graph:



In this example, tracking two separate versions of the variable $a1$ is sufficient to distinguish the objects of type B and C as they are passed through method f, meaning that the analysis can accurately track which version of g is called in each program location.

Call-string context sensitivity has its limits, however. Consider the following example, adapted from notes by Ryder:

```
interface X { void g(); }
class Y implements X { void g() { ... } }
class Z implements X { void g() { ... } }
class A {
        X x;
        void setX(X v) { helper(v)^h; }
        void helper(X vh) { x = vh; }
        X getX() { return x; }
}

// in main()
A a1 = new A();    // allocates Oa1
A a2 = new A();    // allocates Oa2
a1.setX(new Y())^Y; // allocates OY
a2.setX(new Z())^Z; // allocates OZ
X x1 = a1.getX();
X x2 = a2.getX();
```

```
x1.g();          // which g() is called?
x2.g();          // which g() is called?
```

If we analyze this example with a 1-CFA style call-string sensitive pointer analysis, we get the following analysis results:

Context	Variable	Location	Notes
•	a1	Oa1	
•	a2	Oa2	
Y	this	Oa1	
Y	v	OY	
h	this	Oa1	
h	vh	OY	
Oa1	x	OY	
Z	this	Oa2	
Z	v	OZ	
h	this	Oa1,Oa2	updated
h	vh	OY, OZ	updated
Oa1	x	OY, OZ	updated
Oa2	x	OY, OZ	
•	x1	OY, OZ	
•	x1	OY, OZ	

Essentially, because of the helper method, one function call's worth of context sensitivity is insufficient to distinguish the calls to setX and helper for the objects Oa1 and Oa2. We could fix this by increasing context sensitivity, e.g. by going to a 2-CFA analysis that tracks call strings of length two. This has a very high cost in practice, however; 2-CFA does not scale well to large object-oriented programs.

A better solution comes from the insight that in the above example, call-strings are really tracking the wrong kind of context. What we need to do is distinguish between Oa1 and Oa2. In other words, the call chain does not matter so much; we want to be sensitive to the receiver object.

An alternative approach based on this idea is called object-sensitive analysis. It uses for the context not the call site, but rather the receiver object. In this case, we index everything not by a calling point n but instead by a receiver object l. The rules are as follows:

$$\frac{l \vdash p := \mathbf{new}_k\ A}{l_l^k \in p_l}\ \textit{new}$$

$$\frac{l \vdash p := q \quad l_l \in q_l}{l_l \in p_l}\ \textit{copy}$$

$$\frac{l \vdash x.f := y \quad l_x \in x_l \quad l_y \in y_l}{l_y \in l_x.f}\ \textit{field-read}$$

$$\frac{l \vdash x := y.f \quad l_y \in y_l \quad l_z \in l_y.f}{l_z \in x_l}\ \textit{field-assign}$$

$$\frac{l \vdash x.f(y) \quad l_x \in x_l \quad l_y \in y_l \quad [\![f(z) = e]\!] \in \textit{Program}}{l_x \in \mathbf{this}_{l_x} \quad l_y \in z_{l_x} \quad l_x \vdash e}\ \textit{call}$$

Now if we reanalyze the example above, we get:

Context	Variable	Location
•	a1	Oa1
•	a2	Oa2
Oa1	v	OY
Oa1	vh	OY
Oa1	x	OY
Oa2	v	OZ
Oa2	vh	OZ
Oa2	x	OZ
•	x1	OY
•	x1	OZ

In practice, object-sensitive analysis appears to be the best approach to context sensitivity in the pointer or call-graph construction analysis of object-oriented programs. Intuitively, it seems that organizing a program around objects makes the objects themselves the most interesting thing to analyze.

The state of the art implementation technique for points-to analysis of object-oriented programs was presented by Bravenboer and Smaragdakis in OOPSLA 2009. Their approach generates declarative Datalog code to represent the input program, and a datalog evaluation engine solves what

are essentially declarative constraints to get the analysis result.

In an more recent POPL 2011 paper analyzing object-sensitivity, Smaragdakis, Bravenboer, and Lhoták demonstrate that it is more effective than call-string sensitivity. They also propose a technique known as type-sensitive analysis which tracks only the type of the receiver (and, for depths $\geqslant 2$, the type of the object that created the reciever, etc.), and show that type-sensitive analysis is nearly as precise as object-sensitive analysis and much more scalable.

Lecture Notes: Control Flow Analysis for Functional Languages

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

Lecture 10

1 Analysis of Functional Programs

We now consider analysis of functional programs. Consider an idealized functional language, similar to the core of Scheme or ML, based on the lambda calculus. We can define a grammar for expressions in the language as follows:

$$\begin{array}{rcl} e & ::= & \lambda x.e \\ & | & x \\ & | & e_1\ e_2 \\ & | & n \\ & | & e + e \\ & | & ... \end{array}$$

The grammar includes a definition of an anonymous function $\lambda x.e$, where x is the function argument and e is the body of the function. The function can include variables x or function calls $e_1\ e_2$, where e_1 is the function to be invoked and e_2 is passed to that function as an argument. (In an imperative language this would more typically be written $e_1(e_2)$ but we follow the functional convention here). We evaluate a function call $(\lambda x.e)\ v$ with some value v as the argument by substituting the argument v for all occurrences of x in the body e of the function. For example, $(\lambda x.x + 1)\ 3$ evaluates to $3 + 1$, which of course evaluates further to 4.

A more interesting execution example would be $(\lambda f.f\ 3)(\lambda x.x+1)$. This first evaluates by substituting the argument for f, yielding $(\lambda x.x\ +\ 1)\ 3$. Then we invoke the function getting $3 + 1$ which again evaluates to 4.

Let us consider an analysis such as constant propagation applied to this language. Because functional languages are not based on statements but rather expressions, it is appropriate to consider not just the values of variables, but also the values that expressions evaluate to. We can consider each expression to be labeled with a label $l \in \mathit{Lab}$. Our analysis information σ, then, maps each variable and label to a lattice value. The definition for constant propagation is as follows:

$$\sigma \quad \in \quad \mathit{Var} \cup \mathit{Lab} \to L$$

$$L \quad = \quad \mathbb{Z} + \top$$

We can now define our analysis by defining inference rules that generate constraints which are later solved:

$$\frac{}{[\![n]\!]^l \hookrightarrow \alpha(n) \sqsubseteq \sigma(l)}\ \mathit{const}$$

$$\frac{}{[\![x]\!]^l \hookrightarrow \sigma(x) \sqsubseteq \sigma(l)}\ \mathit{var}$$

In the rules above the constant or variable value flows to the program location l. The rule for function calls is a bit more complex, though. In a functional language, functions are passed around as first-class values, and so it is not always obvious which function we are calling. Although it is not obvious, we still need some way to figure it out, because the value a function returns (which we may hope to track through constant propagation analysis) will inevitably depend on which function is called, as well as the arguments.

The consequence of this is, to do a good job of constant propagation—or, in fact, any program analysis at all—in a functional programming language, we must determine what function(s) may be called at each application in the program. Doing this is called *control flow analysis*.

In order to perform control flow analysis alongside constant propagation, we extend our lattice as follows:

$$L \quad = \quad \mathbb{Z} + \top + \mathcal{P}(\lambda x.e)$$

Thus the analysis information at any given program point, or for any program variable, may be an integer n, or $\top$, or a set of functions that could be stored in the variable or computed at that program point. We can now generate and use this information with the following rules for function definitions and applications:

$$\frac{[\![e]\!]^{l_0} \hookrightarrow C}{[\![\lambda x.e^{l_0}]\!]^{l} \hookrightarrow \{\lambda x.e\} \sqsubseteq \sigma(l) \cup C} \textit{ lambda}$$

$$\frac{[\![e_1]\!]^{l_1} \hookrightarrow C_1 \quad [\![e_2]\!]^{l_2} \hookrightarrow C_2}{[\![e_1^{l_1}\ e_2^{l_2}]\!]^{l} \hookrightarrow C_1 \cup C_2 \cup \forall \lambda x.e_0^{l_0} \in \sigma(l_1) : \sigma(l_2) \sqsubseteq \sigma(x) \wedge \sigma(l_0) \sqsubseteq \sigma(l)} \textit{ apply}$$

The first rule just states that if a literal function is declared at a program location l, that function is part of the lattice value $\sigma(l)$ computed by the analysis for that location. Because we want to analyze the data flow inside the function, we also generate a set of constraints C from the function body and return those constraints as well.

The rule for application first analyzes the function and the argument to extract two sets of constraints C_1 and C_2. We then generate a conditional constraint, saying that for every literal function $\lambda x.e_0$ that the analysis (eventually) determines the function may evaluate to, we must generate additional constraints capture value flow from the formal function argument to the actual argument variable, and from the function result to the calling expression.

Let us consider analysis of the second example program given above. We start by labeling each subexpression as follows: $((\lambda f.(f^a\ 3^b)^c)^e(\lambda x.(x^g + 1^h)^i)^j)^k$. We can now apply the rules one by one to analyze the program:

Var ∪ *Lab*	L	by rule
e	$\lambda f.f\ 3$	lambda
j	$\lambda x.x+1$	lambda
f	$\lambda x.x+1$	apply
a	$\lambda x.x+1$	var
b	3	const
x	3	apply
g	3	var
h	1	const
i	4	add
c	4	apply
k	4	apply

2 m-Calling Context Sensitive Control Flow Analysis (m-CFA)

The simple control flow analysis described above—known as 0-CFA, where CFA stands for Control Flow Analysis and the 0 indicates context insensitivity—works well for simple programs like the example above, but it quickly becomes imprecise in more interesting programs that reuse functions in several calling contexts. The following code illustrates the problem:

let *add* $= \lambda x.\ \lambda y.\ x + y$
let *add5* $= (add\ 5)^{a5}$
let *add6* $= (add\ 6)^{a6}$
let *main* $= (add5\ 2)^{m}$

This example illustrates the functional programming idea of *currying*, in which a function such as *add* that takes two arguments x and y in sequence can be called with only one argument (e.g. 5 in the call labeled $a5$), resulting in a function that can later be called with the second argument (in this case, 2 at the call labeled m). The value 5 for the first argument in this example is stored with the function in the *closure add5*. Thus when the second argument is passed to *add5*, the closure holds the value of x so that the sum $x + y = 5 + 2 = 7$ can be computed.

The use of closures complicates program analysis. In this case, we create two closures, *add5* and *add6*, within the program, binding 5 and 6 and the respective values for x. But unfortunately the program analysis cannot distinguish these two closures, because it only computes one value for x,

and since two different values are passed in, we learn only that x has the value $\top$. This is illustrated in the following analysis. The trace we give below has been shortened to focus only on the variables (the actual analysis, of course, would compute information for each program point too):

Var $\cup$ *Lab*	L	notes
add	$\lambda x.\ \lambda y.\ x + y$	
x	5	when analyzing first call
add5	$\lambda y.\ x + y$	
x	$\top$	when analyzing second call
add6	$\lambda y.\ x + y$	
main	$\top$	

We can add precision using a context-sensitive analysis. One could, in principle, use either the functional or call-string approach to context sensitivity, as described earlier. However, in practice the call-string approach seems to be used for control-flow analysis in functional programming languages, perhaps because in the functional approach there could be many, many contexts for each function, and it is easier to place a bound on the analysis in the call-string approach.

We will add context sensitivity by making our analysis information σ track information separately for different call strings, denoted by Δ. Here a call string is a sequence of labels, each one denoting a function call site, where the sequence can be of any length between 0 and some bound m (in practice m will be in the range 0-2 for scalability reasons):

$$\begin{aligned} \sigma &\in (\mathit{Var} \cup \mathit{Lab}) \times \Delta \to L \\ \Delta &= \mathit{Lab}^{n \leqslant m} \\ L &= \mathbb{Z} + \top + \mathcal{P}((\lambda x.e, \delta)) \end{aligned}$$

When a lambda expression is analyzed, we now consider as part of the lattice the call string context δ in which its free variables were captured.

We can then define a set of rules that generate constraints which, when solved, provide an answer to control-flow analysis, as well as (in this case) constant propagation:

$$\frac{}{\delta \vdash [\![n]\!]^l \hookrightarrow \alpha(n) \sqsubseteq \sigma(l, \delta)} \textit{ const}$$

$$\frac{}{\delta \vdash [\![x]\!]^l \hookrightarrow \sigma(x, \delta) \sqsubseteq \sigma(l, \delta)} \textit{ var}$$

$$\frac{}{\delta \vdash [\![\lambda x.e^{l_0}]\!]^l \hookrightarrow \{(\lambda x.e, \delta)\} \sqsubseteq \sigma(l, \delta)} \textit{ lambda}$$

$$\frac{\begin{array}{l}\delta \vdash [\![e_1]\!]^{l_1} \hookrightarrow C_1 \qquad \delta \vdash [\![e_2]\!]^{l_2} \hookrightarrow C_2 \qquad \delta' = \textit{suffix}(\delta{+}{+}l, m) \\ C_3 = \bigcup_{(\lambda x.e_0^{l_0}, \delta_0) \in \sigma(l_1, \delta)} \sigma(l_2, \delta) \sqsubseteq \sigma(x, \delta') \wedge \sigma(l_0, \delta') \sqsubseteq \sigma(l, \delta) \\ \qquad\qquad \wedge \forall y \in FV(\lambda x.e_0) : \sigma(y, \delta_0) \sqsubseteq \sigma(y, \delta') \\ C_4 = \bigcup_{(\lambda x.e_0^{l_0}, \delta_0) \in \sigma(l_1, \delta)} C \textit{ where } \delta' \vdash [\![e_0]\!]^{l_0} \hookrightarrow C\end{array}}{\delta \vdash [\![e_1^{l_1}\ e_2^{l_2}]\!]^l \hookrightarrow C_1 \cup C_2 \cup C_3 \cup C_4} \textit{ apply}$$

These rules contain a call string context δ in which the analysis of each line of code is done. The rules *const* and *var* are unchanged except for indexing σ by the current context δ. The *lambda* rule now captures the context δ along with the lambda expression, so that when the lamda expression is called the analysis knows in which context to look up the free variables.

Finally, the apply rule has gotten more complicated. A new context δ is formed by appending the current call site l to the old call string, then taking the suffix of length m (or less). We now consider all functions that may be called, as eventually determined by the analysis (our notation is slightly loose here, because the quantifier must be evaluated continuously for more matches as the analysis goes along). For each of these, we produce constraints capturing the flow of values from the formal to actual arguments, and from the result expression to the calling expression. We also produce constraints that bind the free variables in the new context: all free variables in the called function flow from the point δ_0 at which the closure was captured. Finally, in C_4 we collect the constraints that we get from analyzing each of the potentially called functions in the new context δ'.

A final technical note: because the apply rule results in analysis of the called function, if there are recursive calls the derivation may be infinite. Thus we interpret the rules coinductively.

We can now reanalyze the earlier example, observing the benefit of context sensitivity. In the table below, $\bullet$ denotes the empty calling context (e.g. when analyzing the *main* procedure):

Var / Lab, δ	L	notes
add, $\bullet$	$(\lambda x.\ \lambda y.\ x + y,\ \bullet)$	
x, a5	5	
add5, $\bullet$	$(\lambda y.\ x + y,\ a5)$	
x, a6	6	
add6, $\bullet$	$(\lambda y.\ x + y,\ a6)$	
main, $\bullet$	7	

Note three points about this analysis. First, we can distinguish the values of x in the two calling contexts: x is 5 in the context a5 but it is 6 in the context a6. Second, the closures returned to the variables *add5* and *add6* record the scope in which the free variable x was bound when the closure was captured. This means, third, that when we invoke the closure *add5* at program point m, we will know that x was captured in calling context a5, and so when the analysis analyzes the addition, it knows that x holds the constant 5 in this context. This enables constant propagation to compute a precise answer, learning that the variable *main* holds the value 7.

3 Uniform k-Calling Context Sensitive Control Flow Analysis (k-CFA)

m-CFA was proposed recently by Might, Smaragdakis, and Van Horn as a more scalable version of the original k-CFA analysis developed by Shivers for Scheme. While m-CFA now seems to be a better tradeoff between scalability and precision, k-CFA is interesting both for historical reasons and because it illustrates a more precise approach to tracking the values of variables bound in a closure.

The following exmaple illustrates a situation in which m-DFA may be too imprecise:

$$
\begin{array}{ll}
\textbf{let}\ \mathit{adde} & = \lambda x. \\
 & \quad \textbf{let}\ h = \lambda y.\ \lambda z.\ x + y + z \\
 & \quad \textbf{let}\ r = h\ 8 \\
 & \quad \textbf{in}\ r \\
\textbf{let}\ t & = (\mathit{adde}\ 2)^t \\
\textbf{let}\ f & = (\mathit{adde}\ 4)^f \\
\textbf{let}\ e & = (t\ 1)^e
\end{array}
$$

When we analyze it with m-CFA, we get the following results:

Var / *Lab*, δ	L	notes
adde, $\bullet$	$(\lambda x..., \bullet)$	
x, t	2	
y, r	8	
x, r	2	when analyzing first call
t, $\bullet$	$(\lambda z.\ x + y + z,\ r)$	
x, f	4	
x, r	$\top$	when analyzing second call
f, $\bullet$	$(\lambda z.\ x + y + z,\ r)$	
t, $\bullet$	$\top$	

The k-CFA analysis is like m-CFA, except that rather than keeping track of the scope in which a closure was captured, the analysis keeps track of the scope in which each variable captured in the closure was defined. We use an environment η to track this. Note that since η can represent a separately calling context for each variable, rather than merely a single context for all variables, it has the potential to be more accurate, but also much more expensive. We can represent the analysis information as follows:

$$\begin{aligned} \sigma &\in (\mathit{Var} \cup \mathit{Lab}) \times \Delta \to L \\ \Delta &= \mathit{Lab}^{n \leqslant k} \\ L &= \mathbb{Z} + \top + \mathcal{P}(\lambda x.e, \eta) \\ \eta &\in \mathit{Var} \to \Delta \end{aligned}$$

Let us briefly analyze the complexity of this analysis. In the worst case, if a closure captures n different variables, we may have a different call string for each of them. There are $O(n^k)$ different call strings for a program of size n, so if we keep track of one for each of n variables, we have $O(n^{n*k})$ different representations of the contexts for the variables captured in each closure. This exponential blowup is why k-CFA scales so badly. m-CFA is comparatively cheap—there are "only" $O(n^k)$ different contexts for the variables captured in each closure—still expoential in k, but polynomial in n for a fixed (and generally small) k.

We can now define the rules for k-CFA. They are similar to the rules for m-CFA, except that we now have two contexts: the calling context δ, and the environment context η tracking the context in which each variable is bound. When we analyze a variable x, we look it up not in the current

context δ, but the context $\eta(x)$ in which it was bound. When a lambda is analyzed, we track the current environment η with the lambda, as this is the information necessary to determine where captured variables are bound. The application rule is actually somewhat simpler, because we do not copy bound variables into the context of the called procedure:

$$\frac{}{\delta, \eta \vdash [\![n]\!]^l \hookrightarrow \alpha(n) \sqsubseteq \sigma(l, \delta)} \textit{ const}$$

$$\frac{}{\delta, \eta \vdash [\![x]\!]^l \hookrightarrow \sigma(x, \eta(x)) \sqsubseteq \sigma(l, \delta)} \textit{ var}$$

$$\frac{}{\delta, \eta \vdash [\![\lambda x.e^{l_0}]\!]^l \hookrightarrow \{(\lambda x.e, \eta)\} \sqsubseteq \sigma(l, \delta)} \textit{ lambda}$$

$$\frac{\begin{array}{l}\delta, \eta \vdash [\![e_1]\!]^{l_1} \hookrightarrow C_1 \qquad \delta, \eta \vdash [\![e_2]\!]^{l_2} \hookrightarrow C_2 \qquad \delta' = \textit{suffix}(\delta{+}{+}l, k) \\ C_3 = \bigcup_{(\lambda x.e_0^{l_0}, \eta_0) \in \sigma(l_1, \delta)} \sigma(l_2, \delta) \sqsubseteq \sigma(x, \delta') \wedge \sigma(l_0, \delta') \sqsubseteq \sigma(l, \delta) \\ C_4 = \bigcup_{(\lambda x.e_0^{l_0}, \eta_0) \in \sigma(l_1, \delta)} C \textit{ where } \delta', \eta_0 \vdash [\![e_0]\!]^{l_0} \hookrightarrow C\end{array}}{\delta, \eta \vdash [\![e_1^{l_1}\ e_2^{l_2}]\!]^l \hookrightarrow C_1 \cup C_2 \cup C_3 \cup C_4} \textit{ apply}$$

Now we can see how k-CFA analysis can more precisely analyze the latest example program. In the simulation below, we give two tables: one showing the order in which the functions are analyzed, along with the calling context δ and the environment η for each analysis, and the other as usual showing the analysis information computed for the variables in the program:

function	δ	η
main	$\bullet$	$\varnothing$
adde	t	$\{x \mapsto t\}$
h	r	$\{x \mapsto t,\ y \mapsto r\}$
adde	f	$\{x \mapsto f\}$
h	r	$\{x \mapsto f,\ y \mapsto r\}$
$\lambda z....$	e	$\{x \mapsto t,\ y \mapsto r,\ z \mapsto e\}$

Var / Lab, δ	L	notes
adde, $\bullet$	$(\lambda x..., \bullet)$	
x, t	2	
y, r	8	
t, $\bullet$	$(\lambda z.\ x + y + z,\ \{x \mapsto t,\ y \mapsto r\})$	
x, f	4	
f, $\bullet$	$(\lambda z.\ x + y + z,\ \{x \mapsto f,\ y \mapsto r\})$	
z, e	1	
t, $\bullet$	11	

Tracking the definition point of each variable separately is enough to restore precision in this program. However, programs with this structure—in which analysis of the program depends on different calling contexts for bound variables even when the context is the same for the fucntion eventually called—appear to be rare in practice. Might et al. observed no examples among the real programs they tested in which k-CFA was more accurate than m-CFA—but k-CFA was often far more costly. Thus at this point the m-CFA analysis seems to be a better tradeoff between efficiency and precision, compared to k-CFA.

Lecture Notes:
Object-Oriented Call Graph Construction

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

Lecture 11

1 Class Hierarchy Analysis

Analyzing object-oriented programs is challenging in much the same way that analyzing functional programs is challenging: it is not obvious which function is called at a given call site. In order to construct a precise call graph, an analysis must determine what the type of the receiver object is at each call site. Therefore, object-oriented call graph construction algorithms must simultaneously build a call graph and compute aliasing information describing to which objects (and thereby implicity to which types) each variable could point.

The simplest approach is *class hierarchy analysis*. This analysis uses the type of a variable, together with the class hierarchy, to determine what types of object the variable could point to. Unsurprisingly, this analysis is very imprecise, but it can be computed very efficiently: the analysis takes $O(n * t)$ time, because it visits n call sites and at each call site traverses a subtree of size t of the class hierarchy.

2 Rapid Type Analysis

An improvement to class hierarchy analysis is *rapid type analysis*, which eliminates from the hierarchy classes that are never instantiated. The analysis iteratively builds a set of instantiated types, method names invoked, and concrete methods called. Initially, it assumes that `main` is the only concrete method that is called, and that no objects are instantiated. It then

analyzes concrete methods known to be called one by one. When a method name is invoked, it is added to the list, and all concrete methods with that name defined within (or inherited by) types known to be instantiated are added to the called list. When an object is instantiated, its type is added to the list of instantiated types, and all its concrete methods that have a method name that is invoked are added to the called list. This proceeds iteratively until a fixed point is reached, at which point the analysis knows all of the object types that may actually be created at run time.

Rapid type analysis can be considerably more precise than class hierarchy analysis in programs that use libraries that define many types, only a few of which are used by the program. It remains extremely efficient, because it only needs to traverse the program once (in $O(n)$ time) and then build the call graph by visiting each of n call sites and considering a subtree of size t of the class hierarchy, for a total of $O(n * t)$ time.

3 0-CFA Style Object-Oriented Call Graph Construction

Object-oriented call graphs can also be constructed using a pointer analysis such as Andersen's algorithm, either context-insensitive or context-sensitive. The context-sensitive versions are called k-CFA by analogy with control-flow analysis for functional programs. The context-insensitive version is called 0-CFA for the same reason. Essentially, the analysis proceeds as in Andersen's algorithm, but the call graph is built up incrementally as the analysis discovers the types of the objects to which each variable in the program can point.

Even 0-CFA analysis can be considerably more precise than Rapid Type Analysis. For example, in the program below, RTA would assume that any implementation of foo() could be invoked at any program location, but O-CFA can distinguish the two call sites:

```
class A { A foo(A x) { return x; } }
class B extends A { A foo(A x) { return new D(); } }
class D extends A { A foo(A x) { return new A(); } }
class C extends A { A foo(A x) { return this; } }

// in main()
A x = new A();
while (...)
```

```
    x = x.foo(new B()); // may call A.foo, B.foo, or D.foo
A y = new C();
y.foo(x);               // only calls C.foo
```

Lecture Notes: Hoare Logic

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

Revised March 2013

1 Hoare Logic

The goal of Hoare logic is to provide a formal system for reasoning about program correctness. Hoare logic is based on the idea of a specification as a contract between the implementation of a function and its clients. The specification is made up of a precondition and a postcondition. The precondition is a predicate describing the condition the function relies on for correct operation; the client must fulfill this condition. The postcondition is a predicate describing the condition the function establishes after correctly running; the client can rely on this condition being true after the call to the function.

The implementation of a function is *partially correct* with respect to its specification if, assuming the precondition is true just before the function executes, then if the function terminates, the postcondition is true. The implementation is *totally correct* if, again assuming the precondition is true before function executes, the function is guaranteed to terminate and when it does, the postcondition is true. Thus total correctness is partial correctness plus termination.

Note that if a client calls a function without fulfilling its precondition, the function can behave in any way at all and still be correct. Therefore, if it is intended that a function be robust to errors, the precondition should include the possibility of erroneous input and the postcondition should describe what should happen in case of that input (e.g. a specific exception being thrown).

Hoare logic uses Hoare Triples to reason about program correctness. A Hoare Triple is of the form $\{P\}\ S\ \{Q\}$, where P is the precondition, Q is the

postcondition, and S is the statement(s) that implement the function. The (total correctness) meaning of a triple $\{P\}\ S\ \{Q\}$ is that if we start in a state where P is true and execute S, then S will terminate in a state where Q is true.

Consider the Hoare triple $\{x = 5\}\ x := x*2\ \{x > 0\}$. This triple is clearly correct, because if $x = 5$ and we multiply x by 2, we get $x = 10$ which clearly implies that $x > 0$. However, although correct, this Hoare triple is not a precise as we might like. Specifically, we could write a stronger postcondition, i.e. one that implies $x > 0$. For example, $x > 5 \wedge x < 20$ is stronger because it is more informative; it pins down the value of x more precisely than $x > 0$. The strongest postcondition possible is $x = 10$; this is the most useful postcondition. Formally, if $\{P\}\ S\ \{Q\}$ and for all Q such that $\{P\}\ S\ \{Q\}$, $Q \Rightarrow Q$, then Q is the strongest postcondition of S with respect to P.

We can compute the strongest postcondition for a given statement and precondition using the function $sp(S, P)$. Consider the case of a statement of the form $x := E$. If the condition P held before the statement, we now know that P still holds and that $x = E$—where P and E are now in terms of the old, pre-assigned value of x. For example, if P is $x + y = 5$, and S is $x := x + z$, then we should know that $x' + y = 5$ and $x = x' + z$, where x' is the old value of x. The program semantics doesn't keep track of the old value of x, but we can express it by introducing a fresh, existentially quantified variable x'. This gives us the following strongest postcondition for assignment:

$$sp(x := E, P) \quad = \exists x'.[x'/x]P \wedge x = [x'/x]E$$

As described in earlier lectures, the operation $[x'/x]E$ denotes the capture-avoiding substitution of x′ for x in E; we rename bound variables as we do the substitution so as to avoid conflicts.

While this scheme is workable, it is awkward to existentially quantify over a fresh variable at every statement; the formulas produced become unnecessarily complicated, and if we want to use automated theorem provers, the additional quantification tends to cause problems. Dijkstra proposed reasoning instead in terms of *weakest preconditions*, which turns out to work more clearly. If $\{P\}\ S\ \{Q\}$ and for all P such that $\{P\}\ S\ \{Q\}$, $P \Rightarrow P$, then P is the weakest precondition $wp(S, Q)$ of S with respect to Q.

We can define a function yielding the weakest precondition with respect to some postcondition for assignments, statement sequences, and if statements, as follows:

$$
\begin{aligned}
wp(x := E, P) &= [E/x]P \\
wp(S;\ T, Q) &= wp(S,\ wp(T, Q)) \\
wp(\text{if } B \text{ then } S \text{ else } T, Q) &= B \Rightarrow wp(S, Q) \wedge \neg B \Rightarrow wp(T, Q)
\end{aligned}
$$

Verifying loop statements of the form while b do S is more difficult. Because it may not be obvious how many times the loop will execute—and it may in fact be impossible even to bound the number of executions—we cannot mechanically generate a weakest precondition in the style above. Instead, we must reason about the loop inductively.

Intuitively, any loop that is trying to compute a result must operate by establishing that result one step at a time. What we need is an inductive proof that shows that each time the loop executes, we get one step closer to the final result—and that when the loop is complete (i.e. the loop condition is false) that result has been obtained. This reasoning style requires us to write down what we have computed so far after an arbitrary number of iterations; this will serve as an induction hypothesis. The literature calls this form of induction hypothesis a *loop invariant*, because it represents a condition that is always true (i.e. invariant) before and after each execution of the loop.

In order to be used in an inductive proof of a loop, a loop invariant must fulfill the following conditions:

- $P \Rightarrow I$: The invariant is initially true. This condition is necessary as a base case, to establish the induction hypothesis.
- $\{Inv \wedge B\}\ S\ \{Inv\}$: Each execution of the loop preserves the invariant. This isn the inductive case of the proof.
- $(Inv \wedge \neg B) \Rightarrow Q$: The invariant and the loop exit condition imply the postcondition. This condition is simply demonstrating that the induction hypothesis/loop invariant we have chosen is sufficiently strong to prove our postcondition S.

The procedure outlined above only verifies partial correctness, because it does not reason about how many times the loop may execute. In order to verify full correctness, we must place an upper bound on the number of remaining times the loop body will execute. This bound is typically called a *variant function*, written v, because it is variant: we must prove that it decreases each time we go through the loop. If we can also prove that

whenever the bound v is equal to (or less than) zero, the loop condition will be false, then we have verified that the loop will terminate.

More formally, we must come up with an integer-valued variant function v that fulfils the following conditions:

- $Inv \wedge v \leq 0 \Rightarrow \neg B$: If we are entering the loop body (i.e. the loop condition B evaluates to true) and the invariant holds, then v must be strictly positive.
- $\{Inv \wedge B \wedge v = V\}\ S\ \{v < V\}$: The value of the variant function decreases each time the loop body executes (here V is a placeholder constant representing the "old" value of v).

2 Proofs with Hoare Logic

Consider the WHILE program used in the previous lecture notes:

$$
\begin{array}{l}
r := 1; \\
i := 0; \\
\text{while } i < m \text{ do} \\
\qquad r := r * n; \\
\qquad i := i + 1
\end{array}
$$

We wish to prove that this function computes the nth power of m and leaves the result in r. We can state this with the postcondition $r = n^m$.

Next, we need to determine the proper precondition. We cannot simply compute it with wp because we do not yet know what the right loop invariant is—and in fact, different loop invariants could lead to different preconditions. However, a bit of reasoning will help. We must have $m \geq 0$ because we have no provision for dividing by n, and we avoid the problematic computation of 0^0 by assuming $n > 0$. Thus our precondition will be $m \geq 0 \wedge n > 0$.

We need to choose a loop invariant. A good hueristic for choosing a loop invariant is often to modify the postcondition of the loop to make it depend on the loop index instead of some other variable. Since the loop index runs from i to m, we can guess that we should replace m with i in the postcondition $r = n^m$. This gives us a first guess that the loop invariant should include $r = n^i$.

This loop invariant is not strong enough (doesn't have enough information), however, because the loop invariant conjoined with the loop exit condition should imply the postcondition. The loop exit condition is $i \geq m$,

but we need to know that $i = m$. We can get this if we add $i \leq m$ to the loop invariant. In addition, for proving the loop body correct, we will also need to add $0 \leq i$ and $n > 0$ to the loop invariant. Thus our full loop invariant will be $r = n^i \wedge 0 \leq i \leq m \wedge n > 0$.

In order to prove full correctness, we need to state a variant function for the loop that can be used to show that the loop will terminate. In this case $m - i$ is a natural choice, because it is positive at each entry to the loop and decreases with each loop iteration.

Our next task is to use weakest preconditions to generate proof obligations that will verify the correctness of the specification. We will first ensure that the invariant is initially true when the loop is reached, by propagating that invariant past the first two statements in the program:

$$\begin{array}{l}\{m \geq 0 \wedge n > 0\} \\ r := 1; \\ i := 0; \\ \{r = n^i \wedge 0 \leq i \leq m \wedge n > 0\}\end{array}$$

We propagate the loop invariant past $i := 0$ to get $r = n^0 \wedge 0 \leq 0 \leq m \wedge n > 0$. We propagate this past $r := 1$ to get $1 = n^0 \wedge 0 \leq 0 \leq m \wedge n > 0$. Thus our proof obligation is to show that:

$$\begin{array}{l}m \geq 0 \wedge n > 0 \\ \Rightarrow 1 = n^0 \wedge 0 \leq 0 \leq m \wedge n > 0\end{array}$$

We prove this with the following logic:

$m \geq 0 \wedge n > 0$	by assumption
$1 = n^0$	because $n^0 = 1$ for all $n > 0$ and we know $n > 0$
$0 \leq 0$	by definition of $\leq$
$0 \leq m$	because $m \geq 0$ by assumption
$n > 0$	by the assumption above
$1 = n^0 \wedge 0 \leq 0 \leq m \wedge n > 0$	by conjunction of the above

We now apply weakest preconditions to the body of the loop. We will first prove the invariant is maintained, then prove the variant function decreases. To show the invariant is preserved, we have:

$$\begin{array}{l}\{r = n^i \wedge 0 \leq i \leq m \wedge n > 0 \wedge i < m\} \\ r := r * n; \\ i := i + 1; \\ \{r = n^i \wedge 0 \leq i \leq m \wedge n > 0\}\end{array}$$

We propagate the invariant past $i := i + 1$ to get $r = n^{i+1} \wedge 0 \leq i + 1 \leq m \wedge n > 0$. We propagate this past $r := r * n$ to get: $r * n = n^{i+1} \wedge 0 \leq i + 1 \leq m \wedge n > 0$. Our proof obligation is therefore:

$$\begin{array}{l} r = n^i \wedge 0 \leq i \leq m \wedge n > 0 \wedge i < m \\ \Rightarrow r * n = n^{i+1} \wedge 0 \leq i + 1 \leq m \wedge n > 0 \end{array}$$

We can prove this as follows:

$r = n^i \wedge 0 \leq i \leq m \wedge n > 0 \wedge i < m$	by assumption
$r * n = n^i * n$	multiplying by n
$r * n = n^{i+1}$	definition of exponentiation
$0 \leq i + 1$	because $0 \leq i$
$i + 1 < m + 1$	by adding 1 to inequality
$i + 1 \leq m$	by definition of $\leq$
$n > 0$	by assumption
$r * n = n^{i+1} \wedge 0 \leq i + 1 \leq m \wedge n > 0$	by conjunction of the above

We have a proof obligation to show that the variant function is positive when we enter the loop. The obligation is to show that the loop invariant and the entry condition imply this:

$$\begin{array}{l} r = n^i \wedge 0 \leq i \leq m \wedge n > 0 \wedge i < m \\ \Rightarrow m - i > 0 \end{array}$$

The proof is trivial:

$r = n^i \wedge 0 \leq i \leq m \wedge n > 0 \wedge i < m$	by assumption
$i < m$	by assumption
$m - i > 0$	subtracting i from both sides

We also need to show that the variant function decreases. We generate the proof obligation using weakest preconditions:

$$\begin{array}{l} \{r = n^i \wedge 0 \leq i \leq m \wedge n > 0 \wedge i < m \wedge m - i = V\} \\ r := r * n; \\ i := i + 1; \\ \{m - i < V\} \end{array}$$

We propagate the condition past $i := i + 1$ to get $m - (i + 1) < V$. Propagating past the next statement has no effect. Our proof obligation is therefore:

$$r = n^i \wedge 0 \leq i \leq m \wedge n > 0 \wedge i < m \wedge m - i = V$$
$$\Rightarrow m - (i + 1) < V$$

Again, the proof is easy:

$r = n^i \wedge 0 \leq i \leq m \wedge n > 0 \wedge i < m \wedge m - i = V$	by assumption
$m - i = V$	by assumption
$m - i - 1 < V$	by definition of $<$
$m - (i + 1) < V$	by arithmetic rules

Last, we need to prove that the postcondition holds when we exit the loop. We have already hinted at why this will be so when we chose the loop invariant. However, we can state the proof obligation formally:

$$r = n^i \wedge 0 \leq i \leq m \wedge n > 0 \wedge i \geq m$$
$$\Rightarrow r = n^m$$

We can prove it as follows:

$r = n^i \wedge 0 \leq i \leq m \wedge n > 0 \wedge i \geq m$	by assumption
$i = m$	because $i \leq m$ and $i \geq m$
$r = n^m$	substituting m for i in assumption