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 $i \in$ program)
     $\sigma[i] = \bot$
$\sigma$[beforeStart] = initialDataflowInformation
worklist = { firstInstruction }

while worklist is not empty
     take an instruction $i$ off the worklist
          var thisInput = $\bot$
          forall (Instruction $j \in$ successors($i$))
                    thisInput = thisInput $\sqcup \sigma[j]$
     let newOutput = flow($i$, thisInput)
          if (newOutput $\neq \sigma[i]$)
                    $\sigma[i]$ = newOutput
                    worklist = worklist $\cup$ 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 $\bot$. 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 ($\bot$). 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. $\square$

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:

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

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.

$$\frac{premise_1 \quad premise_2 \quad \ldots \quad premise_n}{conclusion}$$

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:

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

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

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

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

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

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 $P \vdash c_i \leadsto c_{i+1}$.

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 $0 \leqslant i <$ *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.
- $P \vdash c_i \leadsto c_{i+1}$ 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 $\sqsubseteq$.

$\square$

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.