Lecture Notes: Program Analysis Termination, Correctness, and Optimality

17-355/17-665: Program Analysis *(Spring 2017)*

Jonathan Aldrich
aldrich@cs.cmu.edu

1 Program Analysis Properties

We would like the program analyses we define to have a number of desireable properties. First of all, even though we might be analyzing programs that may have an unbounded set of possible executions, we'd like program analysis to **terminate** in a finite amount of time. Second, we'd like to be able to trust the results of static analysis–i.e. to ensure they are **correct**, so that we can use them for purposes such as program optimization, or so that we can rely on safety guarantees that program analysis can provide. Third, we'd like the results of program analysis to be as precise as possible. While program analysis typically cannot be fully precise without solving the halting problem, we will see that the procedure used in the worklist algorithm guarantees **optimal** results for a given abstraction and set of flow functions.

2 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. 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 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). A 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

All of the analyses we have examined so far have lattices of finite height. However, it is easy to define a lattice that is of infinite height. For example, consider a value set analysis that computes the possible set of integer values that each variable could hold. An element of the lattice is $\sigma_{VS} \in \mathcal{P}^{\mathbb{Z}}$. The ordering between lattice elements is subset. Now we can construct an infinite ascending chain where each element of the chain has an additional integer element that the previous set in the chain did not include. A value analysis defined in a straightforward way will run forever on examples such as the following:

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

Fortunately, we can show that for a lattice of finite height, the worklist algorithm is guaranteed to terminate. We can do so using the following termination metric:

$|worklist| + \sum_{i \in P}(h - \mathit{height}(\mathit{input}[i]))$

where h is the height of the lattice and the *height* function returns the height in the lattice of a particular dataflow value. Consider again the worklist algorithm:

```
**for** Instruction $i \in program$ **do**
    $input[i] = \bot$
**end for**
$input[initial] = initialDataflowValue$
$worklist = initial$

**while** $worklist$ not empty **do**
    remove Instruction $i$ from $worklist$
    $output = flow(i, input[i])$
    **for** Instruction $j \in succ(i)$ **do**
        **if** $output \not\sqsubseteq input[j]$ **then**
            $input[j] = input[j] \sqcup output$
            add $j$ to $worklist$
        **end if**
    **end for**
**end while**
```

In the main loop of the algorithm, we can see that the metric above decreases on each iteration through the loop. First, each iteration removes one element from the worklist, decreasing the metric by one. Of course, at the end of a loop iteration a new node may be added to the worklist—which increases the metric by one—but this is always balanced because the addition only happens if the height of the new dataflow input to the node being added has increased—which correspondingly decreases the metric by one. Since the metric is finite for any lattice that is of finite height, and since it decreases on each iteration through the loop, and since the loop terminates when the metric is zero (because the worklist must be empty in that case), we know the analysis will eventually terminate.

3 Correctness

What does it mean 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. To establish correctness, we will make use of the precise definitions of WHILE3ADDR we gave in the form of operational semantics in the first couple of lectures. We start by formalizing 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 $0 \leqslant i < \mathit{length}(T)$, $\alpha(c_i) \sqsubseteq \sigma_{n_i}$

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 to make 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 $P \vdash c_i \rightsquigarrow c_{i+1}$ implies $\alpha(c_{i+1}) \sqsubseteq f[\![P[n_i]]\!](\alpha(c_i))$

In English: if we take any concrete execution of a program instruction, map the input machine state to the abstract domain using the abstraction function, and apply the flow function, we should get a result that correctly accounts for what happens if we map the actual concrete 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, using 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 overall abstraction function α accepts a complete program configuration (E, n), for zero analysis we can 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 $\alpha(c_{i+1}) \sqsubseteq \sigma_{i+1}$, which finishes the case.

Case $f_Z[\![x := m]\!](\sigma_i) = [x \mapsto N]\sigma_i$ 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 $\alpha(c_{i+1}) \sqsubseteq \sigma_{i+1}$ 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 $f_Z[\![x := y]\!](\sigma) = [x \mapsto \sigma(y)]\sigma$.

3.1 A Critical Property: Monotonicity

Local Soundness captures the intuition that for all possible values that can arise at run time, the flow function for an instruction will produce the correct output. However, one of the key features of static analysis is that it approximates: the analysis results at a given program point may be a conservative approximation (i.e. higher in the lattice) than what we get if we directly abstract any given run time value. We must ensure that this approximation does not lead to problems with local soundness: that even if the analysis computes an approximate result before an instruction, the analysis results afterwards will still be a safe approximation.

To verify this, we need a second property: Monotonicity. Monotonicity checks that a flow function is well-behaved with respect to approximation: if lattice element σ_2 approximates some other element σ_1, then the approximation relationship is preserved by the flow function. More formally:

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

3.2 Montonicity of Zero Analysis

We can formally show that zero analysis is monotone; this is relevant both to the proof of termination, above, and to correctness, next. We will only give a couple of the more interesting cases, and leave the rest as an exercise to the reader:

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 $[x \mapsto \sigma_1(y)]\sigma_1 \sqsubseteq [x \mapsto \sigma_2(y)]\sigma_2$

(α_{simple} and $\sqsubseteq_{simple}$ are simply the unlifted versions of α and $\sqsubseteq$, i.e. they operate on individual values rather than maps.)

Exercise 4. Consider the following (incorrect) flow function for zero analysis:

$$f_Z[\![x := y + 1]\!](\sigma) = \begin{array}{ll} [x \mapsto Z]\sigma & \textit{iff } \sigma(y) = \top \\ [x \mapsto \top]\sigma & \textit{iff } \sigma(y) \neq \top \end{array}$$

(a) Show that the flow function above is not monotonic by providing σ_1 and σ_2 and that the required relationship does not hold after application of the flow function. (b) Give an example of a program for which zero analysis would loop forever if we use a modified version of the worklist algorithm that always merges analysis information from different incoming edges rather than merging new information with the previous information.

Part (b) shows that for a slightly different form of the worklist algorithm, we need monotonicity not just for correctness, but for termination too!

3.3 From Local Soundness to Global Soundness

Now we can show how local soundness and monotonicity can be used together 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 $\sigma_i = \bigsqcup_{j \in \texttt{preds}(i)} f[\![P[j]]\!](\sigma_j)$.

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

Theorem 1 (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.
$f[\![P[n_i]]\!](\sigma_{n_i}) \sqsubseteq \sigma_{n_{i+1}}$ because $\sqcup$ is a least upper bound.
$\alpha(c_{i+1}) \sqsubseteq \sigma_{n_{i+1}}$ 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.

4 Optimality

We would like to prove that the worklist algorithm yields optimal analysis results, relative to the lattice and flow functions we are using. Results from the mathematical theory of lattices allow us to do so.

We can build a "program lattice" as a tuple with one lattice element per program point. Consider an abstraction of the worklist algorithm's inner loop that nondeterministically chooses one statement, applies the flow function to it, and updates the program lattice with the result. This abstraction can be viewed as a function mapping one element of the program lattice to another. It's easy to show that this "analysis function" is monotonic if the original flow functions were monotonic, and that the program lattice is a complete lattice if the original lattice was complete.

We can now apply interesting theory. The Kleene fixed-point theorem states:

Theorem (Kleene Fixed-Point Theorem). Let $(L, \sqsubseteq)$ be a CPO (complete partial order), and let $f : L \to L$ be a Scott-continuous (and therefore monotone) function. Then f has a least fixed point, which is the supremum of the ascending Kleene chain of f.

A complete lattice is a CPO. Furthermore, Scott-continuity is a slightly stronger version of monotonicity: it states that for all subsets S of the lattice, $\sqcup f(S) = f(\sqcup S)$ – that is, if you take the supremum of the set S and apply the function, you get the same thing as applying the function to each element and then taking the supremum of the results. Most reasonable monotonic flow functions you will write are also Scott-continuous.

How does this theory help us? Well, an ascending Kleene chain is just the chain that starts with $\bot$ and is formed by repeated applications of the flow function f. That is, we have the chain:
$\bot \sqsubseteq f(\bot) \sqsubseteq f(f(\bot)) \sqsubseteq \ldots \sqsubseteq f^n(\bot) \sqsubseteq \ldots$

The Kleene fixed-point theorem tells us that we can get the least fixed point—i.e. the most precise analysis result, since places lower in the lattice are more precise—by starting with $\bot$ and repeatedly applying the flow functions. That is exactly what the worklist algorithm does. Thus, the worklist algorithm produces an optimal analysis result for the abstractions represented by a given lattice and set of flow functions.

Notably, a dual theorem can be used the show the existance of a *greatest* fixed point, which we can get by starting with $\top$ everywhere in the program (e.g. on loop back edges, not just at the entry to a procedure) and iterating in the same way. Of course, the greatest fixed point is the most approximate fixed point, so we always prefer the least fixed point because it is more precise.

This discussion leads naturally into a fuller treatment of abstract interpretation, which we will turn to in subsequent lectures/readings.

Acknowledgements

I thank Claire Le Goues for greatly appreciated extensions and refinements to these notes.