Lecture Notes:
Widening Operators and Collecting Semantics

17-355/17-665/17-819: Program Analysis (Spring 2020)
Claire Le Goues*
clegoues@cs.cmu.edu

1 A Collecting Semantics for Reaching Definitions

The approach to dataflow analysis correctness outlined in the previous lectures generalizes naturally when we have a lattice that can be directly abstracted from program configurations c from our execution semantics. Sometimes, however, it would be useful to track other kinds of information, that we cannot get directly from a particular state in program execution. For example, consider *reaching definitions*, which we discussed as an example analysis last week. Although we can track which definitions reach a line using the previously-outlined approach, we cannot see *where* the variables used in an instruction I were last defined.

To solve this problem, we can augment our semantics with additional information that captures the required information. For example, for reaching definitions, we want 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.

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

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

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

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

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

*These notes were developed together with Jonathan Aldrich

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:

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

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 can be tricky for some analyses, but it can be done with a little thought. For example, 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.

2 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:

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

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 $[\infty, -\infty]$.

Note also that this lattice is defined to capture the range of a single variable. As usual, we can lift it to a map from variables to interval lattice elements. Thus we (again) 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:

$$\begin{array}{lll} f_I[\![x := y + z]\!](\sigma) = \sigma[x \mapsto [l, h]] & \text{where } l = \sigma(y).low +_\infty \sigma(z).low \\ & \text{and } h = \sigma(y).high +_\infty \sigma(z).high \\ f_I[\![x := y + z]\!](\sigma) = \sigma & \text{where } \sigma(y) = \bot \vee \sigma(z) = \bot \end{array}$$

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 above-defined lattice, and what consequences does this have for our analysis in practice?*

3 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*. To motivate the widening operator, consider applying interval analysis to the program below:

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

Using the worklist algorithm (strongly connected components first), gives us:

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]	$\top$
...			

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 $\bot, [0,0], [0,\infty], [0,\infty], ...$ 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 analyzing instruction 2 produces:

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, the result should also be an ascending chain. 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:

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

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 $[-\infty, 1]$. The analysis stabilizes after one more iteration.

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.