1 Iterative analysis

Last time we saw that a dataflow analysis can be characterized as a four-tuple $(D, L, \sqcap, F)$: the direction of analysis D, the space of values L, transfer functions F_n, and a meet operator $\sqcap$. We're not yet guaranteed that the worklist algorithm works, however.

Let's consider a simpler algorithm that computes the answer to a dataflow analysis. If the dataflow analysis framework satisfies certain properties to be identified, this algorithm will compute the same thing as the worklist algorithm, but less efficiently. We can think of the worklist algorithm as an optimized version of this iterative analysis algorithm, which avoids recomputing $out(n)$ for nodes n whose value couldn't have changed (because it hasn't changed for any predecessors of n).

Iterative analysis (forward):

- for all n, $out(n) := \top$
- repeat until no change:
  - $in(n) := \bigsqcap_{n' < n} out(n')$
  - $out(n) := F_n(in(n))$

The algorithm updates $out(n)$ for all n on each iteration. If we imagine each of the nodes n as having one of the distinct indices $1, \ldots, N$, we can think of all the values $out(n)$ as forming an N-tuple $(out(n_1), \ldots, out(n_N))$, which is an element of the set L^N.

We can think of the action of each iteration of the loop as mapping an element of L^N to a new element of L^N; that is, it is a function $F : L^N \to L^N$. The action of the algorithm produces a series of tuples until the same tuple happens on two consecutive iterations:

$$
\begin{array}{rl}
& (\top, \top, \ldots, \top) \\
\longrightarrow & (l_1^1, l_2^1, \ldots, l_N^1) \\
\longrightarrow & (l_1^2, l_2^2, \ldots, l_N^2) \\
& \vdots \\
\longrightarrow & (l_1^k, l_2^k, \ldots, l_N^k) \\
\longrightarrow & (l_1^k, l_2^k, \ldots, l_N^k)
\end{array}
$$

- When is this algorithm guaranteed to terminate, i.e., converge on a tuple, and how big can the iteration count k be?
- When does it produce a solution to the dataflow equations?
- When does it produce the *best* solution to the equations?

To get answers to these questions, we need to understand the theory of partial orders, because we will want the space of dataflow values L to be a partial order.

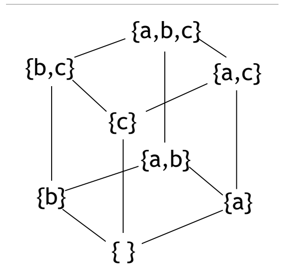

Figure 1: Hasse diagram

2 Partial orders

A *partial order* (or *partially ordered set*, or *poset*) is a set of elements (called the *carrier* of the partial order) along with a relation $\sqsubseteq$ that is:

- reflexive: $x \sqsubseteq x$ for all x.
- transitive: if $x \sqsubseteq y$ and $y \sqsubseteq z$, then $x \sqsubseteq z$.
- antisymmetric: if $x \sqsubseteq y$ and $y \sqsubseteq x$, then x and y are the same element.

The key thing that makes this a *partial* order is that it is possible for two elements to be incomparable; they are not related in either direction.

For dataflow analysis, we interpret the ordering $l_1 \sqsubseteq l_2$ to means that l_2 is a better or more informative result.

Some examples of partial orders are the integers ordered by $\leq$ (i.e., $(\mathbb{Z}, \leq)$), types ordered by the subtyping relation $\leq$ (in many languages), sets ordered by $\subseteq$ (or $\supseteq$), booleans ordered by $\Rightarrow$. If $(L, \sqsubseteq)$ is a partial order, the *dual partial order* $(L, \sqsupseteq)$ is too. Some examples of non-partial orders are the reals ordered by $<$ and pairs of integers ordered by their sums.

2.1 Hasse diagram

A useful way to visualize a partial order is through a Hasse diagram, as shown in Figure 1. This is a diagram for the subsets of $\{a, b, c\}$ with the ordering relation $\subseteq$. In the diagram, elements that are ordered are connected by a line if there is no intermediate element that lies between them in the ordering. And elements connected by a line are displaced vertically to show which is the greater in the relation. Therefore, any two related elements are connected by a path that goes consistently upward or downward in the diagram.

The height of a partial order is the number of edges n on the longest chain of elements $l_0 \sqsubseteq l_1 \sqsubseteq l_2 \sqsubseteq \ldots l_n$. Therefore, the height of the example in Figure 1 is 3.

2.2 Lattices

A *lower bound* of two elements x and y is an element that is less than both of them. Some partial orders have the property that every two elements have a *greatest lower bound*, or *GLB*, or *meet*. It is written $x \sqcap y$, and pronounced as "x meet y".

The meet of two elements is above all other lower bounds in the ordering: $z \sqsubseteq x \wedge z \sqsubseteq y \Rightarrow z \sqsubseteq x \sqcap y$.

Dually, for some partial orders, every two elements have a greatest upper bound (LUB), written $x \sqcup y$ and pronounced "x join y".

If a partial order has both a meet and a join for every pair of elements, it is called a *lattice*. If it has a meet for every pair of elements, it is a *lower semilattice*. If it has a join for every pair of elements, it is an *upper semilattice*. We will be interested only in meets, so we will be working with lower semilattices, which we may simply abbreviate to "lattice" (and most of the partial orders we care about are, in fact, full lattices).

2.3 Tuples

Suppose that L is a partial order. Then the set of tuples L^N is also a partial order under the *componentwise ordering*:

$$(l_1, l_2, \ldots, l_N) \sqsubseteq (l'_1, l'_2, \ldots, l'_N) \iff \forall_{i \in 1..N}\ l_i \sqsubseteq l'_i$$

You can check for yourself that if L is a partial order, this ordering on L^N is also reflexive, transitive, and antisymmetric.

If L is a lattice, then L^N is also a lattice, with the meet (or join) taken componentwise:

$$(l_1, \ldots, l_N) \sqcap (l'_1, \ldots, l'_N) = (l_1 \sqcap l'_1, \ldots, l_N \sqcap l'_N)$$
$$(l_1, \ldots, l_N) \sqcup (l'_1, \ldots, l'_N) = (l_1 \sqcup l'_1, \ldots, l_N \sqcup l'_N)$$

To see that this works for meets, we need to show that $(l_1 \sqcap l'_1, \ldots, l_N \sqcap l'_N)$ is greater than any other lower bound for $(l_1, \ldots, l_N)$ and $(l'_1, \ldots, l'_N)$. Suppose we have such a lower bound $(l''_1, \ldots, l''_N)$. Since it is a lower bound, for all i, $l''_i \sqsubseteq l_i$ and also $l''_i \sqsubseteq l'_i$. But that implies that $l''_i \sqsubseteq l_i \sqcap l'_i$. Therefore, according to the componentwise ordering on L^N, $(l''_1, \ldots, l''_N) \sqsubseteq (l_1 \sqcap l'_1, \ldots, l_N \sqcap l'_N)$.

3 Monotonicity

The iterative analysis algorithm starts from the top of the lattice L^N, $(\top, \top, \ldots, \top)$, and repeatedly applies a function $F : L^N \to L^N$ to it, until a *fixed point* of the function is reached: a tuple $X = (l^k_1, \ldots, l^k_N)$ such that $F(X) = X$. As the algorithm executes, a series of tuples $X_0, X_1, X_2, \ldots, X_k$ is produced, where $X_0 = (\top, \top, \ldots, \top)$ and X_k is the fixed point of F.

Given that a fixed point is reached, all the dataflow equations must be satisfied; otherwise, a different tuple would have resulted from the last iteration of the loop. So if the algorithm terminates, it does find a solution. How do we know that it finds a solution?

The key is to observe that the transfer functions F_n are normally *monotonic*, and therefore the function F is too. A function on a partial order is monotonic if it preserves ordering:

Monotonicity: A function $f : L \to L$ is monotonic if $x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$.

In the context of dataflow analysis, monotonicity makes sense. We can think about the transfer functions F_n as describing what we know after a node executes, given what we know beforehand. Having more information before the node executes should not cause us to have less information afterward; it should only help or at worst have no benefit.

The function F is constructed out of the transfer functions F_n and the meet operator. If the transfer functions are monotonic on L, the function F is monotonic on L^N. To see why, let us first check that the meet operator is monotonic.

Theorem 1 (The meet operator is monotonic on its arguments)

$$x \sqsubseteq y \Rightarrow x \sqcap z \sqsubseteq y \sqcap z$$

This proposition is depicted in Figure 2. Since the ordering $\sqsubseteq$ is transitive, we know that $x \sqcap z \sqsubseteq y$. This means $x \sqcap z$ is a lower bound for both y and z, and therefore, it is bounded above by the greatest lower bound for y and z, which is $y \sqcap z$.

The function F is formed by the composition of monotonic transfer functions and the monotonic meet operator, as depicted in Figure 3, so it is also monotonic.

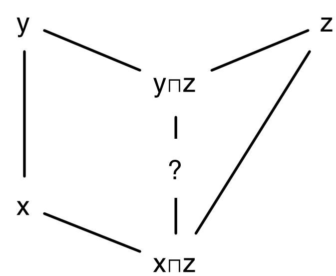

Figure 2: Monotonicity of meet

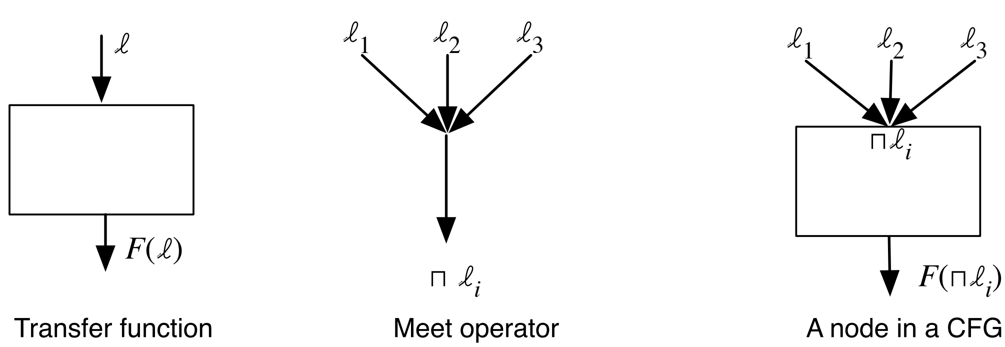

Figure 3: Dataflow analysis framework components

4 Termination

Iterative analysis starts with top element X_0 and applies F to it. The result, which we called X_1, must be ordered with respect to X_0; that is, $X_1 \sqsubseteq X_0$. Because F is monotonic, $F(X_1) \sqsubseteq F(X_0)$; that is, $X_2 \sqsubseteq X_1$. This pattern must continue: for all n, $X_{n+1} \sqsubseteq X_n$, which we can see by induction. If we assume that $X_n \sqsubseteq X_{n-1}$, then by monotonicity of F, $X_{n+1} \sqsubseteq X_n$. Therefore the successive dataflow values produced by the algorithm form a chain of distinct elements:

$$X_k \sqsubseteq X_{k-1} \sqsubseteq \ldots \sqsubseteq X_2 \sqsubseteq X_1 \sqsubseteq X_0$$

If the lattice L^N has infinite height, there is no guarantee that this chain won't continue indefinitely. But for most of the problems we care about, the lattice L has some finite height (call it h). Therefore, the lattice of tuples L^N has height at most Nh. Once the iterative analysis algorithm has run Nh iterations, it must have arrived at the bottom of the chain: convergence is achieved in k iterations where $k \leq Nh$.

5 Example: live variable analysis

In live variable analysis, the dataflow values are sets of live variables. We want to find as *few* variables live as possible to enable the most optimization, so the ordering $\sqsubseteq$ is $\supseteq$, the top element $\top$ is $\emptyset$, and the meet operator $\sqcap$ is $\cup$.

Are the transfer functions monotonic? Recall that:

$$F_n(l) = use(n) \cup (l - def(n))$$

So if $l \sqsubseteq l'$, then $l \supseteq l'$. Suppose we have an element $x \in F_n(l') = use(n) \cup (l' - def(n))$. Then either $x \in use(n)$, or else $x \in l' - def(n)$, in which case $x \in l - def(n)$. In either case $x \in F_n(l)$. Since this is true for arbitrary x, $F_n(l) \supseteq F_n(l')$, as required.

6 The meet-over-all-paths solution and distributivity

We know that we get a solution to the dataflow equations if we run iterative analysis. But is it the best possible solution? For example, in live variable analysis, we defined a variable as live if there is *any path* leading from the current program point where that variable will be used. The set of live variables is therefore the union (i.e., the meet) over all possible paths of the variables that are live along any of the paths. In most dataflow analyses, like this one, we are trying to arrive at the meet-over-all-paths (MOP) solution:

$$out(n) = \bigsqcap_{\text{all paths } p_0 p_1 \ldots p_k n} F_n(F_{p_k}(F_{p_{k-1}}(\ldots(F_{p_1}(\ldots(F_{p_0}(\top)))))))$$

The reason we might not get the MOP solution is that even if that our transfer functions capture perfect reasoning, there is still the possibility of losing information whenever we take a meet. If meet *doesn't* lose information, then we should get the same answer to the dataflow analysis when we duplicate the subsequent node, perform the analysis on the replicas, and then recombine the results using meet (see Figure 4).

If this is true, we say that the transfer functions are *distributive*, and we can pull a meet operation out from the argument to F_n and take it after applying F_n:

$$F_n(l_1 \sqcap l_2) = F_n(l_1) \sqcap F_n(l_2)$$

What the iterative analysis computes is an alternating application of meet and transfer functions (the updates to $in(n)$ and $out(n)$, respectively), so the result is something like this:

$$out(n) = F_n\left(\bigsqcap_{n' \prec n} F_{n'}\left(\bigsqcap_{n'' \prec n'} F_{n''}\left(\bigsqcap_{n''' \prec n''} \ldots\right)\right)\right)$$

But if the F_n's are all distributive, that means we can pull out all the meets, giving us exactly the MOP solution.

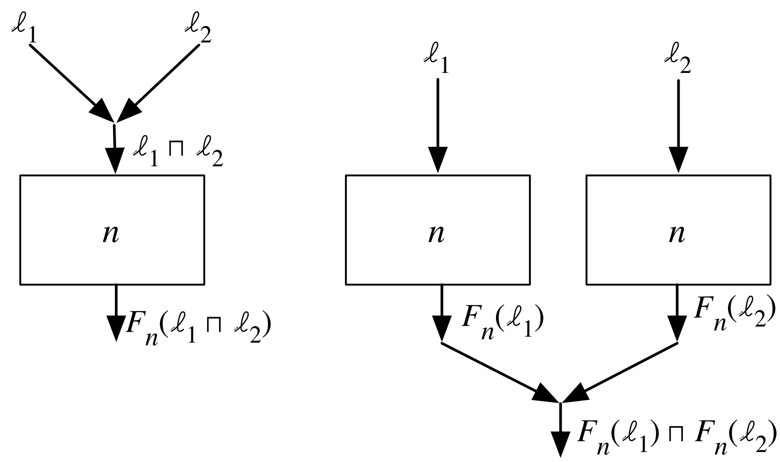

Figure 4: Analyses that are equivalent if meet loses no information

7 Example: live variable analysis

Does live variable analysis give us the MOP solution? Yes, which we can see by showing that F_n is distributive:

$$\begin{aligned} F_n(x \sqcap y) &= use(n) \cup ((x \cup y) - def(n)) \\ &= use(n) \cup ((x - def(n)) \cup (y - def(n))) \\ &= (use(n) \cup (x - def(n))) \cup (use(n) \cup (y - def(n))) \\ &= F_n(x) \sqcap F_n(y) \end{aligned}$$

8 Example: constant propagation

In "classic" constant propagation, the dataflow value is a mapping from variables to either a constant value c, the "don't know" value $\perp$, or the "no assignment yet" value $\top$, with $\perp \sqsubseteq c \sqsubseteq \top$ for all c. For a node $z = x\,OP\,y$, then, we compute the outgoing value of x as follows (? represents any value):

x	y	z
c_1	c_2	$c_1\,OP\,c_2$
$\perp$	?	$\perp$
?	$\perp$	$\perp$
$\top$	?	$\top$
?	$\top$	$\top$

The transfer function is not distributive. Consider a node that computes $z = x + y$ and has two predecessor nodes with output values $\{x \mapsto 2, y \mapsto 3\}$ and $\{x \mapsto 3, y \mapsto 2\}$. The meet of these values is $\{x \mapsto \perp, y \mapsto \perp\}$, so the node will compute $\{x \mapsto \perp, y \mapsto k, z \mapsto \perp\}$. However, applying the transfer function to the individual values yields $\{x \mapsto 2, y \mapsto 3, z \mapsto 5\}$ and $\{x \mapsto 3, y \mapsto 2, z \mapsto 5\}$, and the resulting meet is $\{x \mapsto \perp, y \mapsto \perp, z \mapsto 5\}$. Thus, information is lost by taking the meet before applying the transfer function.