Notes on Dataflow Analysis

Andrew Tolmach

February 8, 2024

1 Dataflow Analysis and Lattices

The general framework of a dataflow analysis problem can be set up as follows. Suppose we have a program and its control-flow graph (CFG). The nodes of the CFG can be individual instructions or basic blocks (straight-line sequences of instructions); it doesn't much matter. Our goal is to calculate some information I about the program state immediately before and after execution of the code associated with each CFG node; that is, we want compute $I_{in}[i]$ and $I_{out}[i]$ for each node i. For liveness analysis, I is the set of variables live at the specified point; for constant analysis, I might be a map from variable names to values; more on this in later sections. Our plan is to do this calculation gradually, by repeatedly refining an *approximation* to the correct information. At any given point in the calculation, the set I might be such an approximation; eventually, we want it to converge to the true value.

The nice thing is that the dataflow framework doesn't care what the type I is, as long as values of this type form a *lattice* (or more precisely, a join semi-lattice). Rather than recapitulate the generic formal definition of a lattice (Wikipedia is your friend!), I'll just describe the requirements we'll put on I in the context of dataflow analysis, where it is useful to think in terms of the "information content" of each value of I.

- We must define a partial order $\sqsubseteq$ on the elements of I, where $x \sqsubseteq y$ means "x is no better defined than y" or "y is at least as good an approximation as x."
- There is a "bottom" element in I, written $\bot$, such that $\bot \sqsubseteq x$ for every element $x \in I$. $\bot$ represents having no information at all. We'll typically start the analysis assuming that I_{in} and I_{out} are $\bot$ at most or all nodes.
- For any two elements $x, y \in S$, the least upper bound of x and y with respect to $\sqsubseteq$ exists and is in S. We call this the *join* of x and y, written $x \sqcup y$, and it represents the result of merging the information in x and y, as we will need to do at "join points" in the CFG. (The fact that we model "join points" with lattice "joins" is mostly just a happy coincidence.)

We also define a *transfer function* $tr : i \times I \to I$ that describes how executing the code at a node changes the information we know. The transfer function together with the join operator describe how to perform a single round of improvement in the approximation we're computing. Exactly how this works depends on the *direction* of the dataflow analysis. In a *forward* analysis, the transfer function computes $I_{out}[i]$ from $I_{in}[i]$, and the framework compute $I_{in}[i]$ as the join of $I_{out}[j]$ for all predecessors j of i. In a *backward* analysis, the transfer function computes $I_{in}[i]$ from $I_{out}[i]$, and

the framework compute $I_{out}[i]$ as the join of $I_{in}[k]$ for all successors k of i. (Liveness is a backward analysis, as you know.) Unsurprisingly, a backward analysis is just the same as a forward analysis on the transpose of the CFG, so the same algorithm will do for both.

So a single round in the approximation process consists of performing a suitable join and then applying tr at every node. We can view the effect of this as a function f that tranforms one approximation (a complete collection of, say, I_{out} sets for each node) to a new approximation (another complete collection of I_{out} sets for each node). To be a little more precise, we can think of f as a function of type $M \to M$, where M is the mapping from i to $I_{out}[i]$. It turns out that we can make M into a lattice as well, by defining:

- $m_1 \sqsubseteq m_2$ iff $m_i[i] \sqsubseteq m_2[i]$ for all i
- $\bot_M[i] = \bot$ for all i
- $(m_1 \sqcup m_2)[i] = m_1[i] \sqcup m_2[i]$ for all i

How will we know when we have performed enough iterations? Answer: when applying f ceases to change the mapping. In other words, a solution m_s of our problem is one that is a *fixed point* of f, i.e. $f(m_s) = m_s$. For problems (including liveness) where we are creating *under*-approximations, we want the *least* fixed point, i.e. a fixed point in which the I sets have only the elements they absolutely must, and no extraneous junk. It turns out that some general theory on lattices (the "Kleene fixed-point theorem") says that, under some suitable restrictions on f (it must be "monotone") and the lattice (it must have no "infinite ascending chains"), such a fixed point will always be reached by repeatedly applying f to $\bot_M$.

Fig. 6.5 of the book gives an efficient iterative algorithm for solving an arbitrary dataflow problem posed in this framework, by iterating f in this way. Rather than completely recomputing the information sets on each approximation round, it uses a worklist to focus only those nodes whose inputs have changed, but the result is the same. Essentially the same code is available to you as function `analyze_dataflow` in the file `dataflow_analysis.py`. To use this code, you just need to define functions for tr and $\sqcup$ and give the value of $\bot$.

2 Liveness Analysis as an instance of Dataflow Analysis

Let's see how the general dataflow analysis framework can be instantiated to do liveness analysis.

Here the information $I = LIVE$ we're computing at each node is a *set* of live variables, so we want to define a lattice whose elements are such sets. This is easy: we take $\sqsubseteq$ to be ordinary set $\subseteq$, $\bot$ to be the empty set ($\emptyset$), and $\sqcup$ to be set union ($\cup$). (In principle, we need to check that set union really does compute least upper bounds with respect to inclusion, but this is fairly trivial to see.)

We want this to be a backward analysis, and so the transfer function should compute $LIVE_{in}[i]$ from $LIVE_{out}[i]$. So we just plug in the equations we already developed for liveness:

$tr(L) = (L - write[i]) \cup read[i]$

and the framework takes care of calculating $LIVE_{out}[i]$ from the $LIVE_{in}$ sets of the sucesssors of i using $\sqcup = \cup$.

Now we just apply the `analyze_dataflow` function to this choice of tr, $\bot$, and $\sqcup$. (And note that tr for basic blocks is already implemented as `calc_liveness`.) That's all there is to it!

Well, almost. As described here, you might think that *tr* is a pure function, with no side-effects. And indeed this works as far as computing a solution goes. But the `analyze_dataflow` function doesn't return anything! So if we want to actually record the solution (and of course we do!) that needs to be done as a side-effect within the *tr* function. This is conveniently done by making *tr* a nested local function which can access the program's blocks and update a solution map defined in an outer function. Also, you'll want to have some special-case code in *tr* to handle the `Label("conclusion")` node, which doesn't have defining block.

3 Constant Propagation as an Instance of Dataflow Analysis

As another application of dataflow analysis, consider the problem of computing, at each program instruction, a "known constant" map $K : \mathit{Var} \to \mathit{Val}$ from variables $x \in \mathit{Var}$ to their known (integer) values $v \in \mathit{Val}$. Of course, in general we don't know the value of a variable at compile time, but we *do* know it when the variable has been assigned to a literal constant and not subsequently re-assigned. (We could also extend this to the case when the variable is assigned to an expression whose own variables all have known values, but for simplicity we'll ignore this extension here.) As usual, the main difficulty comes from join points in the control flow, where the known values on the incoming branches may be different. Also, at the start of execution, the value of every variable is undefined. To handle these issues, we define the codomain *Val* of the known constant map to be $\mathbb{Z} \cup \{\bot, \top\}$, where $\mathbb{Z}$ is the set of integers, $\bot$ is a special value meaning "unknown', and $\top$ is a new special value meaning "not a constant." We can visualize the lattice like this:

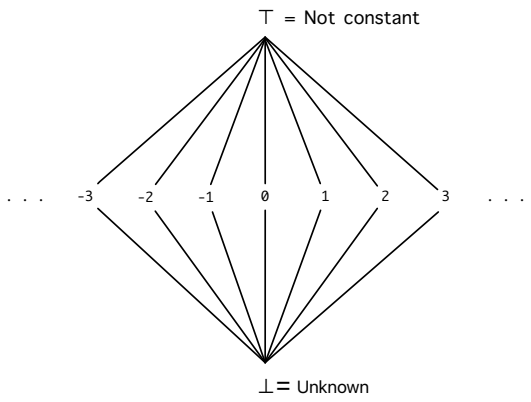

Figure 1 contains an example showing how the map evolves over a simple program. Note that the map can mostly be computed in a single forward pass, but, as usual, the presence of a loop (the backward jump to `L3`) means that we must be a little cleverer; hence the need for employing dataflow analysis, in this case a *forward* analysis.

To fit this problem into the dataflow framework, we take $I = K = \mathit{Var} \to \mathit{Val}$. We need to define a lattice on K. First we define the $\sqsubseteq$ relation on *Val*:

- $v \sqsubseteq v$ for all values v (including integers i, $\bot$, and $\top$)
- $\bot \sqsubseteq i \sqsubseteq \top$ for all integers i

and a corresponding join relation $\sqcup$:

- $v \sqcup v = v$ for all values v (including integers i, $\bot$, and $\top$)

```
            { x:⊥, y:⊥, z:⊥ }
      x = 1
            { x:1, y:⊥, z:⊥ }
      y = 2
            { x:1, y:2, z:⊥ }
      if ... goto L1
            { x:1, y:2, z:⊥ }
      y = 4
            { x:1, y:4, z:⊥ }
      z = 3
            { x:1, y:4, z:3 }
      goto L2

L1:         { x:1, y:2, z:⊥ }
      z = 3
            { x:1, y:2, z:3 }
      goto L2

L2:         { x:1, y:⊤, z:3 }   # note y was changed in one branch;
      goto L3                    # z was updated to the same thing in both branches

L3:         { x:⊤, y:⊤, z:3 }   # note x is updated in the loop
     if ... goto L4
     y = 2
            { x:⊤, y:2, z:3 }
     x = x + 1
            { x:⊤, y:2, z:3 }
     goto L3

L4:         { x:⊤, y:⊤, z:3 }
```

Figure 1: Example of constant propagation

- $i \sqcup j = \top$ for all integers $i \neq j$
- $i \sqcup \bot = \bot \sqcup i = i$ for all integers i
- $v \sqcup \top = \top \sqcup v = \top$ for all values v (including integers i and $\bot$)

Now we can define a lattice structure on the maps $k \in K$ as follows.

- $k_1 \sqsubseteq k_2$ iff $k_1(x) \subseteq k_2(x)$ for all $x \in \mathit{Var}$
- $(k_1 \sqcup k_2)(x) = k_1(x) \sqcup k_2(x)$ for all $x \in \mathit{Var}$
- $\bot(x) = \bot$ for all $x \in Var$

Since this is a forward analysis, the transfer function for instruction *instr* must compute the post-instruction map $K_{out}[\mathit{instr}]$ from the pre-instruction map $K_{in}[\mathit{instr}]$. It can be given as

$tr(k)(x)$ `=`
`  if` *instr* `= ‘‘`x `= `i`’’ then` i `(where` i `is an integer literal)`
`  else if` *instr* `= ‘‘`x `= ...’’ then` $\top$
`  else` $k(x)$

Similarly to before, the framework takes care of calculating $K_{in}[\mathit{instr}]$ from the K_{out} sets of the predecessors of *instr* using our definition of $\sqcup$. At join points, this has the effect of forcing the value associated with any variable x to be $\top$ unless it has the same (known) value on each incoming CFG edge. Again, we just need to apply the `analyze_dataflow` function with this choice of tr, $\bot$, and $\sqcup$ in order to compute the K_{in} sets at each instruction.