Iterative Solution of Data Flow Problems

This algorithm will use DFO numbering to determine the order in which blocks are visited for evaluation. We iterate over the nodes until convergence.

EvalDF{
   For (all \( n \in \text{CFG} \)) {
      \( \text{soln}(n) = T \)
      \( \text{ReEval}(n) = \text{true} \)
   }
   Repeat
   LoopAgain = false
   For (all \( n \in \text{CFG} \) in DFO order) {
      If (ReEval(n)) {
         \( \text{ReEval}(n) = \text{false} \)
         OldSoln = \( \text{soln}(n) \)
         \( \text{In} = \bigwedge_{p \in \text{Pred}(n)} \text{soln}(p) \)
         \( \text{soln}(n) = f_n(\text{In}) \)
         If (\( \text{soln}(n) \neq \text{OldSoln} \)) {
            For (all \( s \in \text{Succ}(n) \)) {
               \( \text{ReEval}(s) = \text{true} \)
               LoopAgain = LoopAgain OR IsBackEdge(n,s)
            }
         }
      }
   } Until (! LoopAgain)
}

Example: Reaching Definitions

We'll do this as a set-valued problem (though it really is just three bit-valued analyses, since each analysis is independent).

\( L \) is the power set of Basic Blocks
\( \bigwedge \) is set union
\( T \) is \( \emptyset \); \( \bot \) is the set of all blocks
\( a \leq b \equiv b \subseteq a \)
\( f_3(\text{in}) = \{3\} \)
\( f_6(\text{in}) = \{6\} \)
\( f_7(\text{in}) = \{7\} \)
For all other blocks, \( f_b(\text{in}) = \text{in} \)
We'll track soln and ReEval across multiple passes

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>Loop-Again</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial</td>
<td>φ</td>
<td>φ</td>
<td>φ</td>
<td>φ</td>
<td>φ</td>
<td>φ</td>
<td>φ</td>
<td>φ</td>
<td>φ</td>
<td>true</td>
</tr>
<tr>
<td>Pass 1</td>
<td>φ</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>(3)</td>
<td>(3)</td>
<td>(6)</td>
<td>(7)</td>
<td>(7)</td>
<td>true</td>
</tr>
<tr>
<td>Pass 2</td>
<td>φ</td>
<td>(3)</td>
<td>(3)</td>
<td>(3,7)</td>
<td>(3,7)</td>
<td>(6)</td>
<td>(7)</td>
<td>(7)</td>
<td></td>
<td>true</td>
</tr>
<tr>
<td>Pass 3</td>
<td>φ</td>
<td>(3,7)</td>
<td>(3,7)</td>
<td>(3,7)</td>
<td>(3,7)</td>
<td>(6)</td>
<td>(7)</td>
<td>(7)</td>
<td></td>
<td>false</td>
</tr>
</tbody>
</table>

Properties of Iterative Data Flow Analysis

- If the height of the lattice (the maximum distance from T to ⊥) is finite, then termination is guaranteed. Why?
- Recall that transfer functions are assumed monotone (a ≤ b ⇒ f(a) ≤ f(b)). Also, ∧ has the property that a ∧ b ≤ a and a ∧ b ≤ b.
- At each iteration, some solution value must change, else we halt. If something changes it must “move down” the lattice (we start at T). If the lattice has finite height, each block’s value can change only a bounded number of times. Hence termination is guaranteed.

How Many Iterations are Needed?

- If the iterative data flow algorithm terminates, a valid solution must have been computed. (This is because data flow values flow forward, and any change along a backedge forces another iteration.)
- Can we bound the number of iterations needed to compute a data flow solution?
- In our example, 3 passes were needed, but why?
- In an “ideal” CFG, with no loops or backedges, only 1 pass is needed. With backedges, it can take several passes for a value computed in one block to reach a block that depends upon the value.
Let \( p \) be the maximum number of backedges in any acyclic path in the CFG.

Then \((p+1)\) passes suffice to propagate a data flow value to any other block that uses it.

Recall that any block's value can change only a bounded number of times. In fact, the height of the lattice (maximum distance from top to bottom) is that bound.

Thus the maximum number of passes in our iterative data flow evaluator = 

\[(p+1) \times \text{Height of Lattice}\]

In our example, \( p = 2 \) and lattice height really was 1 (we did 3 independent bit valued problems).

So passes needed = \((2+1)*1 = 3\).

**Rapid Data Flow Frameworks**

We still have the concern that it may take many passes to traverse a solution lattice that has a significant height.

Many data flow problems are rapid. For rapid data flow problems, extra passes to feed back values along cyclic paths aren’t needed.

For a data flow problem to be rapid we require that:

\[
(\forall a \in A)(\forall f \in F) \ a \land f(T) \leq f(a)
\]

This is an odd requirement that states that using \( f(T) \) as a very crude approximation to a value computed by \( F \) is OK when joined using the \( \land \) operator. In effect the term "\( a \)" rather than \( f(T) \) is dominant).

(Recall that \( a \land f(a) \leq f(a) \) always holds.)

**How does the Rapid Data Flow Property Help?**

Consider a direct feedback loop (the idea holds for indirect loops too):

\[
\text{a is an input from outside the loop.} \\
\text{Our concern is how often we'll need to reevaluate } f, \text{ as new values are computed and fed back into } f. \\
\text{Initially, we'll use } T \text{ to model the value on the backedge.}
\]
Iteration 1: Input = a ∧ T = a
Output = f(a)
Iteration 2: Input = a ∧ f(a)
Output = f(a ∧ f(a))
Iteration 3: Input = a ∧ f(a ∧ f(a))

Now we'll exploit the rapid data flow property: b ∧ f(T) ≤ f(b)
Let b ≡ a ∧ f(a)
Then a ∧ f(a) ∧ f(T) ≤ f(a ∧ f(a)) (*)
Note that x ≤ y ⇒ a ∧ x ≤ a ∧ y (**) 
To prove this, recall that
(1) p ∧ q = p ⇒ p ≤ q
(2) x ≤ y ⇒ x ∧ y = x
Thus (a∧x)∧(a∧y) = a∧(x∧y) = (a∧x)
(by 2) ⇒ (a∧x) ≤ (a∧y) (by 1).

From (*) and (**) we get
a∧a∧f(a)∧f(T) ≤ f(a∧f(a))∧a (***)
Now a ≤ T ⇒ f(a) ≤ f(T) ⇒
f(a)∧f(T) = f(a).
Using this on (***) we get
a∧f(a) ≤ f(a∧f(a))∧a
That is, Input_2 ≤ Input_3
Note too that
a ∧ f(a) ≤ a ⇒ f(a∧f(a)) ≤ f(a) ⇒
a ∧ f(a∧f(a)) ≤ a ∧ f(a)
That is, Input_3 ≤ Input_2
Thus we conclude Input_2 = Input_3,
which means we can stop after two passes independent of lattice height!
(One initial visit plus one reevaluation via the backedge.)

**Many Important Data Flow Problems are Rapid**

Consider reaching definitions, done as sets. We may have many definitions to the same variable, so the height of the lattice may be large.
L is the power set of Basic Blocks ∧ is set union
T is φ; ⊥ is the set of all blocks
a ≤ b ≡ a ⊇ b
f_b(in) = (In − Kill_b) U Gen_b
where Gen_b is the last definition to a variable in b,
Kill_b is all defs to a variable except the last one in b,

Kill_b is empty if there is no def to a variable in b.
The Rapid Data Flow Property is
a ∧ f(T) ≤ f(a)
In terms of Reaching Definitions this is
a U f(φ) ⊇ f(a) ≡
a U (φ − Kill) U Gen ⊇ (a − Kill) U Gen
Simplifying,
a U Gen ⊇ (a − Kill) U Gen
which always holds.
Recall

Here it took two passes to transmit the def in b7 to b1, so we expect 3 passes to evaluate independent of the lattice height.

Constant Propagation isn't Rapid

We require that
\[ a \wedge f(T) \leq f(a) \]
Consider

Look at the transfer function for the second (bottom) block.

\[ f(t) = t' \text{ where} \]
\[ t'(v) = \text{case}(v)\{ \]
\[ k: t(j); \]
\[ j: t(i); \]
\[ i: 2; \} \]
Let \( a = (\bot,1,1) \).
\[ f(T) = (2,T,T) \]
\[ a \wedge f(T) = (\bot,1,1) \wedge (2,T,T) = (\bot,1,1) \]
\[ f(a) = f(\bot,1,1) = (2,\bot,1). \]
Now \((\bot,1,1)\) is not \(\leq (2,\bot,1)\) so this problem isn't rapid.

Let's follow the iterations:
Pass 1: In = \((1,1,1) \wedge (T,T,T) = (1,1,1)\)
\[ \text{Out} = (2,1,1) \]
Pass 2: In = \((1,1,1) \wedge (2,1,1) = (\bot,1,1)\)
\[ \text{Out} = (2,\bot,1) \]
Pass 3: In = \((1,1,1) \wedge (2,\bot,1) = (\bot,\bot,1)\)
\[ \text{Out} = (2,\bot,\bot) \]
This took 3 passes. In general, if we had \(N\) variables, we could require \(N\) passes, with each pass resolving the constant status of one variable.
**How Good Is Iterative Data Flow Analysis?**

A single execution of a program will follow some path
\[ b_0, b_1, b_2, ..., b_n. \]

The Data Flow solution along this path is
\[ f_n(...f_2(f_1(f_0(T))...)) = f(b_0, b_1, ..., b_n). \]

The best possible static data flow solution at some block \( b \) is computed over all possible paths from \( b_0 \) to \( b \).

Let \( P_b = \{ \text{all paths from } b_0 \text{ to } b \} \).

Thus \( \bigwedge_{p \in P_b} f(p) = \text{MOP}(b) \leq f(p_i). \)

This means \( \text{MOP}(b) \) is always a safe approximation to the "true" solution \( f(p_i) \).

If we have the distributive property for transfer functions,
\[ f(a \wedge b) = f(a) \wedge f(b) \]
then our iterative algorithm always computes the MOP solution, the best static solution possible.

To prove this, note that for trivial path of length 1, containing only the start block, \( b_0 \), the algorithm computes \( f_0(T) \) which is \( \text{MOP}(b_0) \) (trivially).

Now assume that the iterative algorithm for paths of length \( n \) or less to block \( c \) does compute \( \text{MOP}(c) \).

We'll show that for paths to block \( b \) of length \( n+1 \), \( \text{MOP}(b) \) is computed.

Let \( P \) be the set of all paths to \( b \) of length \( n+1 \) or less.

The paths in \( P \) end with \( b \).
\[ \text{MOP}(b) = f_b(f(P_1) \wedge f_b(f(P_2) \wedge \ldots) \]

where \( P_1, P_2, \ldots \) are the prefixes (of length \( n \) or less) of paths in \( P \) with \( b \) removed.

Using the distributive property,
\[ f_b(f(P_1) \wedge f_b(f(P_2) \wedge \ldots = f_b(f(P_1) \wedge f(P_2) \wedge \ldots). \]

But note that \( f(P_1) \wedge f(P_2) \wedge \ldots \) is just the input to \( f_b \) in our iterative algorithm, which then applies \( f_b \).

Thus \( \text{MOP}(b) \) for paths of length \( n+1 \) is computed.
For data flow problems that aren't distributive (like constant propagation), the iterative solution is ≤ the MOP solution. This means that the solution is a safe approximation, but perhaps not as "sharp" as we might wish.

**Reading Assignment**

Read “An Efficient Method of Computing Static Single Assignment Form.” (Linked from the class Web page.)

**Exploiting Structure in Data Flow Analysis**

So far we haven’t utilized the fact that CFGs are constructed from standard programming language constructs like IFs, Fors, and Whiles. Instead of iterating across a given CFG, we can isolate, and solve symbolically, subgraphs that correspond to "standard" programming language constructs.

We can then progressively simplify the CFG until we reach a single node, or until we reach a CFG structure that matches no standard pattern.

In the latter case, we can solve the residual graph using our iterative evaluator.

**Three Program-Building Operations**

1. Sequential Execution (";")
2. Conditional Execution (If, Switch)
3. Iterative Execution (While, For, Repeat)
**Sequential Execution**

We can reduce a sequential "chain" of basic blocks:

\[ b_1 \rightarrow b_2 \rightarrow \cdots \rightarrow b_n \]

into a single composite block:

\[ b_{seq} \]

The transfer function of \( b_{seq} \) is

\[ f_{seq} = f_n \circ f_{n-1} \circ \cdots \circ f_1 \]

where \( \circ \) is functional composition.

**Conditional Execution**

Given the basic blocks:

\[ b_1 \rightarrow b_2 \rightarrow \cdots \rightarrow b_n \]

we create a single composite block:

\[ b_{cond} \]

The transfer function of \( b_{cond} \) is

\[ f_{cond} = f_{L1} \circ f_p \wedge f_{L2} \circ f_p \]

**Iterative Execution**

Repeat Loop

Given the basic blocks:

\[ b_B \rightarrow b_C \rightarrow b_B \]

we create a single composite block:

\[ b_{repeat} \]

Here \( b_B \) is the loop body, and \( b_C \) is the loop control.

If the loop iterates once, the transfer function is \( f_C \circ f_B \).

If the loop iterates twice, the transfer function is \( (f_C \circ f_B) \circ (f_C \circ f_B) \).

Considering all paths, the transfer function is \( (f_C \circ f_B) \wedge (f_C \circ f_B)^2 \wedge \cdots \)

Define \( \text{fix } f \equiv f \wedge f^2 \wedge f^3 \wedge \cdots \)

The transfer function of repeat is then

\[ f_{repeat} = \text{fix}(f_C \circ f_B) \]
While Loop.

Given the basic blocks:

\[ b_C \]
\[ b_B \]

we create a single composite block:

\[ b_{\text{while}} \]

Here again \( b_B \) is the loop body, and \( b_C \) is the loop control.
The loop always executes \( b_C \) at least once, and always executes \( b_C \) as the last block before exiting.

Evaluating Fixed Points

For lattices of height \( H \), and monotone transfer functions, \( \text{fix} \ f \) needs to look at no more than \( H \) terms.

In practice, we can give \( \text{fix} \ f \) an operational definition, suitable for implementation:

Evaluate

\[
(\text{fix} \ f)(x) \{ \\
\text{prev} = \text{soln} = f(x); \\
\text{while} \ (\text{prev} \neq \text{new} = f(\text{prev})) \{ \\
\text{prev} = \text{new}; \\
\text{soln} = \text{soln} \land \text{new}; \\
\} \\
\text{return} \ \text{soln}; \\
\}
\]

The transfer function of a while is therefore

\[ f_{\text{while}} = f_C \land \text{fix}(f_C \circ f_B) \circ f_C \]

Example—Reaching Definitions

The transfer functions are either constant-valued \( \{f_1 = \{b_1\}, f_4 = \{b_4\}, f_5 = \{b_5\}\} \) or identity functions \( \{f_2 = f_3 = f_6 = f_7 = \text{Id}\} \).
First we isolate and reduce the conditional:
\[ f_C = f_4 \circ f_3 \land f_5 \circ f_3 = \{b4\} \circ \text{Id} \cup \{b5\} \circ \text{Id} = \{b4,b5\} \]

Substituting, we get

We can combine \(b_C\) and \(b_6\), to get a block equivalent to \(b_C\). That is,
\[ f_6 \circ f_C = \text{Id} \circ f_C = f_C \]

We now have

We isolate and reduce the while loop formed by \(b_2\) and \(b_C\), creating \(b_W\).
The transfer function is
\[
\begin{align*}
    f_W &= f_2 \land (\text{fix}(f_2 \circ f_C)) \circ f_2 = \\
    &= \text{Id} \cup (\text{fix}(\text{Id} \circ f_C)) \circ \text{Id} = \\
    &= \text{Id} \cup (\text{fix}(f_C)) = \\
    &= \text{Id} \cup (f_C \land f_C^2 \land f_C^3 \land \ldots) = \\
    &= \text{Id} \cup \{b4,b5\}
\end{align*}
\]

We now have

We compose these three sequential blocks to get the whole solution, \(f_P\).
\[ f_P = \text{Id} \circ (\text{Id} \cup \{b4,b5\}) \circ \{b1\} = \{b1,b4,b5\}. \]
These are the definitions that reach the end of the program.
We can expand subgraphs to get the solutions at interior blocks.
Thus at the beginning of the while, the solution is \{b1\}.
At the head if the If, the solution is
\[(\text{Id} \cup (\text{Id} \circ f_{c} \circ \text{Id}) \cup (\text{Id} \circ f_{c} \circ \text{Id} \circ f_{c} \circ \text{Id}) \cup \ldots)\circ \{\{b1\}\}\]
= \{b1\} \cup \{b4,b5\} \cup \{b4,b5\} \cup \ldots = \{b1,b4,b5\}
At the head of the then part of the If, the solution is \text{Id(\{b1,b4,b5\})} = \{b1,b4,b5\}.