Overview

Last Lecture

- Introduced two data-flow problems
- Introduced round-robin iterative algorithm
- Reasoned about termination, correctness, and speed
  - I got one slide ahead of myself and made a claim (at the board) about the number of passes that the round-robin algorithm needs
  - We fix that today
Iterative Data-flow Analysis

Correctness

• Does the iterative algorithm compute the desired answer?

Admissible Function Spaces

1. \( \forall f \in F, \forall x, y \in L, f(x \land y) = f(x) \land f(y) \)
2. \( \exists f_i \in F \) such that \( \forall x \in L, f_i(x) = x \)
3. \( f, g \in F \exists h \in F \) such that \( h(x) = f(g(x)) \)
4. \( \forall x \in L, \exists \) a finite subset \( H \subseteq F \) such that \( x = \bigwedge_{f \in H} f(\bot) \)

If \( F \) meets these four conditions, then an instance of the problem will have a unique fixed point solution

\[
\Rightarrow \text{LFP} = \text{MFP} = \text{MOP}
\]

\[
\Rightarrow \text{order of evaluation does not matter}
\]

Both \( \text{DOM} \) & \( \text{LIVE} \) meet all four criteria

If meet does not distribute over function application, then the fixed point solution may not be unique. The iterative algorithm will find a FP.

Iterative Data-flow Analysis

Speed

If a data-flow framework meets those admissibility conditions then it has a unique fixed-point solution

• The iterative algorithm finds the MOP solution — the best answer
• The solution does not depend on order of computation
• Algorithm can choose an order that converges quickly

Intuition

• Choose an order that propagates changes as far as possible on each “sweep”
  > Process a node’s predecessors before the node
• Cycles pose problems, of course
  > Ignore back edges when computing the order?
Ordering the Nodes to Maximize Propagation

- Reverse postorder visits predecessors before visiting a node
- Use reverse postorder on reverse CFG for backward problems
  - Reverse postorder on reverse CFG is not reverse preorder

See exercise 9.4 in EaC2e for an example

Iterative Data-flow Analysis

**Speed**

- For a problem with an admissible function space & a bounded semilattice,
  - If the functions all meet the rapid condition, *i.e.*, 
    \[ \forall f, g \in F, \forall x \in L, f(g(\perp)) \geq g(\perp) \wedge f(x) \wedge x \]
    
    then, a round-robin, reverse-postorder iterative algorithm will halt in \( d(G) + 3 \) passes over a graph \( G \)

\( d(G) \) is the loop-connectedness of the graph w.r.t a DFST
  - Maximal number of back edges in an acyclic path
  - Several studies suggest that, in practice, \( d(G) \) is small \( (<3) \)
  - For most CFGs, \( d(G) \) is independent of the specific DFST

Both DOM & LIVE are rapid frameworks
Iterative Data-flow analysis

What does this mean?

- Reverse postorder
  - Easily computed order that increases propagation per pass
- Round-robin iterative algorithm
  - Visit all the nodes in a consistent order (RPO)
  - Do it again until the sets stop changing
- Rapid condition
  - Most classic global data-flow problems meet this condition

These conditions are easily met

- Admissible framework, rapid function space
- Round-robin, reverse-postorder, iterative algorithm

⇒ The analysis runs in (effectively) linear time

Some problems are not admissible

Global Constant Propagation

S1: \{b=3, c=4\} \quad S2: \{b=1, c=6\}

Function “f” models the block’s effects

- f(S1) = \{a=7, b=3, c=4\}
- f(S2) = \{a=7, b=1, c=6\}
- f(S1 \lor S2) = \emptyset

- First condition in the admissibility criteria
  \forall f \in F, \forall x, y \in L, f(x \land y) = f(x) \land f(y)
- Constant propagation is not admissible
  - Kam & Ullman time bound does not hold
  - There are tight time bounds, however, based on lattice height
  - Require a variable-by-variable formulation …
Classic Constant Propagation

Transformation: Global Constant Folding

- Along every path to \( p \), \( v \) has same known value
- Specialize computation at \( p \) based on \( v \)'s value

Data-flow problem: Constant Propagation

Domain is the set of pairs \( <v_i, c_i> \) where \( v_i \) is a variable and \( c_i \in C \)

\[
\text{CONSTANTS}(b) = \wedge_{p \in \text{preds}(b)} f_p(\text{CONSTANTS}(p))
\]

- \( \wedge \) performs a pairwise meet on two sets of pairs
- \( f_p(x) \) is a block specific function that models the effects of block \( p \) on the \( <v_i, c_i> \) pairs in \( x \)

Constant propagation is a *forward* flow problem

Example

Meet operation requires more explanation

- \( c_1 \wedge c_2 = c_1 \) if \( c_1 = c_2 \), else \( \bot \) \hspace{1cm} (*bottom & top as expected*)

What about \( f_p \)?

- If \( p \) has one statement then
  \[
  x \leftarrow y \quad \text{with } \text{CONSTANTS}(p) = \{<x,l_1>, <y,l_2>, ...\}
  \]
  then \( f_p(\text{CONSTANTS}(p)) = \text{CONSTANTS}(p) - <x,l_1> + <x,l_2> \)
  \[
  x \leftarrow y \text{ op } z \quad \text{with } \text{CONSTANTS}(p) = \{<x,l_1>, <y,l_2>, ..., <z,l_3>, ...\}
  \]
  then \( f_p(\text{CONSTANTS}(p)) = \text{CONSTANTS}(p) - <x,l_1> + <x,l_2 \text{ op } l_3> \)

- If \( p \) has \( n \) statements then
  \[
  f_p(\text{CONSTANTS}(p)) = f_n(f_{n-1}(f_{n-2}(...f_2(f_1(\text{CONSTANTS}(p)))...)))
  \]
  where \( f_i \) is the function generated by the \( i^{th} \) statement in \( p \)

\( f_p \) interprets \( p \) over \( \text{CONSTANTS} \)
Some Admissible Problems Are Not Rapid

Interprocedural May Modify sets

\[ \text{shift}(a,b,c,d,e,f) \]
\[ \{ \]
\[ \text{reference } a,b,c,d,e,f; \]
\[ \text{local } t; \]
\[ \ldots \]
\[ \text{call } \text{shift}(t,a,b,c,d,e); \]
\[ f = 1; \]
\[ \ldots \]
\[ \} \]

- Iterations proportional to number of parameters
  - Not a function of the call graph
  - Can make example arbitrarily bad
- Proportional to length of chain of bindings...

\[ \text{shift} \]

\[ \begin{array}{cccccc}
  a & b & c & d & e & f \\
\end{array} \]

Nothing to do with \( d(G) \)

Classic Algorithm: Round-robin Iterative Algorithm

\[ \text{DOM}(n_0) \leftarrow \{ n_0 \} \]
\[ \text{for } i \leftarrow 1 \text{ to } |N| \]
\[ \quad \text{DOM}(n_i) \leftarrow \{ N \} \]
\[ \text{change } \leftarrow \text{true} \]
\[ \text{while (change)} \]
\[ \quad \text{change } \leftarrow \text{false} \]
\[ \quad \text{for } i \leftarrow 0 \text{ to } |N| \]
\[ \quad \quad \text{TEMP } \leftarrow \{ n_i \} \cup \bigcap_{p \in \text{pred}(n_i)} \text{DOM}(p) \]
\[ \quad \text{if DOM}(n_i) \neq \text{TEMP then} \]
\[ \quad \quad \text{change } \leftarrow \text{true} \]
\[ \quad \quad \text{DOM}(n_i) \leftarrow \text{TEMP} \]

The round-robin algorithm makes a complete sweep over graph
- Not all nodes have new information
- Some of these computations are useless
- A “worklist” formulation can avoid some useless work
**Classic Algorithm: Worklist Iterative Algorithm**

- \( \text{DOM}(n_0) \leftarrow \{ n_0 \} \)
- Worklist \( \leftarrow \emptyset \)

for \( i \leftarrow 1 \) to \( |N| \)
- \( \text{DOM}(n_i) \leftarrow \{ N \} \)
- Worklist \( \leftarrow \text{Worklist} \cup \{ n_i \} \)

while (Worklist \( \neq \emptyset \))
- remove a block \( n \) from Worklist
- TEMP \( \leftarrow \{ n \} \cup \{ \cap_{p \in \text{pred}(n)} \text{DOM}(p) \} \)
- if \( \text{DOM}(n) \neq \text{TEMP} \) then
  - \( \text{DOM}(n) \leftarrow \text{TEMP} \)
  - for each \( s \in \text{succ}(n) \)
    - Worklist \( \leftarrow \text{Worklist} \cup \{ s \} \)

At each stage, Worklist contains nodes whose information is new
- Not all those nodes will have their DOM sets change
- Only those nodes *can* have their DOM sets change

**Round-robin Versus Worklist Algorithm**

- Termination, correctness, & complexity arguments are for the round-robin version
- The worklist algorithm does less work; it visits fewer nodes
- Consider a version with two worklists: *current* & *next*
  - Draw nodes from *current* in RPO; add them to *next*
  - Swap the worklists when *current* becomes empty (1 r-r pass)
  - Makes same changes, in same order, as round robin
  - Termination & correctness arguments should carry over
- Is this the best order? Maybe not …
  - Iterate around a loop until it stabilizes
  - Many data structures for worklist, many orders
  - Your mileage will vary *(measurable effect)*

Worklist algorithm is a sparse version of round-robin algorithm
Classic Algorithm: Round-robin Iterative Algorithm

\[
\begin{align*}
\text{DOM}(n_0) & \leftarrow \{n_0\} \\
\text{Current} & \leftarrow \emptyset \\
\text{Next} & \leftarrow \emptyset \\
\text{for } i & \leftarrow 1 \text{ to } |N| \\
\text{DOM}(n_i) & \leftarrow \{N\} \\
\text{Current} & \leftarrow \text{Current} \cup \{n_i\} \\
\text{while } (\text{Current} \neq \emptyset) \\
\text{remove next block } n & \text{ from Current} \\
\text{TEMP} & \leftarrow \{n\} \cup (\cap_{p \in \text{pred}(n)} \text{DOM}(p)) \\
\text{if } \text{DOM}(n) & \neq \text{TEMP} \\
\text{DOM}(n) & \leftarrow \text{TEMP} \\
\text{for each } s & \in \text{succ}(n) \\
\text{Next} & \leftarrow \text{Next} \cup \{s\} \\
\text{if } \text{Current} = \emptyset & \text{ then} \\
\text{Current} & \leftarrow \text{Next} \\
\text{Next} & \leftarrow \emptyset
\end{align*}
\]

This version of the worklist algorithm sweeps the graph in the same RPO order as the round-robin algorithm.

- Let \(\text{Current} \& \text{Next}\) be priority queues ordered by RPO number
- Imposes a slight added cost in return for guaranteed behavior
- In practice, the simpler version may iterate around a loop more times, but its sparsity makes that expense bearable.

Tradeoff between expense of the priority queue and the extra iterations.

Iterative Data-flow Analysis

How do we use these results?
- Prove that data-flow framework is admissible & rapid
  - Its just algebra
  - Most (but not all) global data-flow problems are rapid
  - This is a property of \(F\)

- Code up an iterative algorithm
  - World’s simplest data-flow algorithm

- Rely on the results
  - Theoretically sound, robust algorithm

This lets us ignore most of the other data-flow algorithms in 512