Data Flow Analysis and Optimizations

Last Time

• Optimizations with SSA

Today

• Data Flow Analysis
• Data Flow Frameworks
• Constant Propagation
• Reaching Definitions

Data Flow Analysis

Data flow analysis tells us things we want to know about programs. For example:

• Is this computation loop invariant?
• Which definition reaches this use?
• Is this value a constant?

Example:

```
if (X)
    X = true
else
    A = 3
    B = 2
    C = A + B
```

Data Flow Framework

1. A semilattice \( \mathcal{L} \) with a binary meet operation \( \sqcap \), such that \( a, b, c \in \mathcal{L} \):
   
   • \( a \sqcap a = a \)  \hspace{2cm}  (idempotent)
   • \( a \sqcap b = b \sqcap a \)  \hspace{2cm}  (commutative)
   • \( a \sqcap (b \sqcap c) = (a \sqcap b) \sqcap c \)  \hspace{2cm}  (associative)

2. \( \sqcap \) imposes an order on \( \mathcal{L} : \forall a, b \in \mathcal{L} \):
   
   • \( a \sqsupseteq b \iff a \sqcap b = b \)
   • \( a \sqsupseteq b \) and \( a \neq b \)

3. A semilattice has a unique bottom element \( \bot : \forall a \in \mathcal{L} \):
   
   • \( a \sqcap \bot = \bot \)
   • \( a \sqsupseteq \bot \)

4. It has a unique top or identity element \( \top : \forall a \in \mathcal{L} \):
   
   • \( a \sqcap \top = a \)
   • \( \top \sqsupseteq a \)

Meet Over All Paths Solution

\[ MOP(v) = \sqcap_{q \in \text{Paths}(v)} T_q(\top) \]
Problem Representation

- Choose a semilattice \( L \) to represent facts
- Attach to each \( a \in L \) a meaning
  each \( a \in L \) is a distinct a set of known facts
- With each node \( n \), associate a function \( f_n : L \rightarrow L \)
  \( f_n \) models behavior of code corresponding to \( n \)
- Let \( F \) be the set of all functions the code generates

Constant Propagation

Constant propagation lattice:

\[
\begin{array}{cccccc}
\vdash & 2 & 1 & 0 & \vdash \\
\vdash & \vdash & \vdash & \vdash & \vdash \\
\vdash & \vdash & \vdash & \vdash & \vdash \\
2 & 1 & 0 & \vdash & \vdash \\
1 & 0 & \vdash & \vdash & \vdash \\
0 & \vdash & \vdash & \vdash & \vdash \\
\vdash & \vdash & \vdash & \vdash & \vdash \\
\end{array}
\]

1. meet rules
   - \( a \sqcap T = a \)
   - \( a \sqcap \bot = \bot \)
   - \( a \sqcap a = a \iff \text{constant } a \text{ and } a = a \)
   - \( a \sqcap b = \bot \text{ otherwise} \)
2. meet properties impose a partial order on \( L \)
   - \( 3 \sqcap 3 = 3 \)
   - \( 3 \sqcap 2 = 2 \sqcap 3 \)
   - \( 3 \sqcap (2 \sqcap 4) = (3 \sqcap 2) \sqcap 4 \)
3. bottom
   - \( a \sqcap \bot = \bot \text{ for every } a \in L \).
   - \( \forall a \in L, a \geq \bot \)
4. top
   - \( a \sqcap T = a \text{ for every } a \in L \)
   - \( \forall a \in L, T \geq a \)

Admissible Function Spaces

For a bounded semilattice \( L, F : L \rightarrow L \) is an admissible function space \( \iff \)

1. Monotonic:
   \[ \forall f \in F, \forall x, y \in L, x \leq y \Rightarrow f(x) \leq f(y) \]
2. Identity operation:
   \[ \exists f_1 \in F : \forall x \in L, f_1(x) = x \]
3. Closed under composition:
   \[ f, g \in F \Rightarrow f \circ g \in F \]
   \[ \text{where } \forall x \in L, [f \circ g](x) = f(g(x)) \]
4. \( \bot \) exists to any \( x \in L \)
   \[ \forall x \in L, \exists \text{ a finite subset } H \subseteq F \ni x = \cap f \in H f(\bot) \]
Monotone Data Flow Framework

is a triple \( \langle \mathcal{L}, \sqcap, \mathcal{F} \rangle \) where

- \( \sqcap \) is the meet operation, or confluence operator
- \( \langle \mathcal{L}, \sqcap \rangle \) is a semilattice of finite length with bottom \( \perp \)
- \( \mathcal{F} \) is a monotone function space on \( \mathcal{L} \):
  a set of unary functions such that each operation \( f \in \mathcal{F} \) is monotonic:
  \[
  \forall f \in \mathcal{F}, \forall x, y \in \mathcal{L}, x \leq y \Rightarrow f(x) \leq f(y)
  \]

A monotone data flow framework \( \langle \mathcal{L}, \sqcap \rangle \) is distributive \( \iff \)

\[
\forall f \in \mathcal{F}, \forall x, y \in \mathcal{L}, f(x \sqcap y) = f(x) \sqcap f(y)
\]

Meet Over All Paths Solution

\[
MOP(v) = \sqcap_{q \in \text{Paths}(v)} T_q(\top)
\]

Constant Propagation

Example Framework

1. Is CP monotonic?
   - Yes

2. Is CP distributive?
   - No

3. Is every solution a meet over all paths solution?
   - No

Reaching Definitions

For each vertex, find the set of variable definitions that might reach that vertex.

\( GEN(v) \) variable \( v \) may be defined or assigned to

\( KILL(v) \) variable \( v \) is defined, overwriting other definitions

```
1: read N
2: call check(N)
3: i <= 1
4: while i < N do
5:   a[i] := a[i] + i
6:   i := i + 1
7: end
8: print a[n]
```

<table>
<thead>
<tr>
<th>GEN</th>
<th>KILL</th>
<th>PRED</th>
<th>SUCC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>N</td>
<td>N</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>N</td>
<td>N2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>i3</td>
<td>i</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td>3.7</td>
</tr>
<tr>
<td>5</td>
<td>a5</td>
<td></td>
<td>4.6</td>
</tr>
<tr>
<td>6</td>
<td>i6</td>
<td>i</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td>6.4</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td>4</td>
</tr>
</tbody>
</table>
Reaching Definitions: Transfer Function

\[ \text{IN}(v) = \bigcup_{p \in \text{PRE}(v)} \text{OUT}(p) \]

\[ \text{OUT}(v) \text{ the set of reaching definitions just after statement } v \]

\[ \text{IN}(v) = \text{GEN}(v) \cup (\text{IN}(v) - \text{KILL}(v)) \]

\text{IN} \text{ is an inherited attribute}

\text{OUT} \text{ is a synthesized attribute}

\text{GEN} \text{ and } \text{KILL} \text{ are basic attributes}

\textbf{Forward} \text{ data flow problems propagate from predecessors of } v \text{ to } v

\textbf{Backward} \text{ data flow problems propagate from successors of } v \text{ to } v

Reaching Definitions

\textbf{Monotone Data Flow Framework}

- \( A \) = set of generations, generation = (statement, variable)
- Lattice: \( \mathcal{L} = \langle 2^A, \cup \rangle \)
- \( 2^A \equiv \) the set of all subsets of \( A \)
- What does this look like?
- initial value = {}
- transfer function \( t_v(x) = \text{GEN}(v) \cup (x - \text{KILL}(v)) \)
- monotone: \( x \subseteq y \Rightarrow t_v(x) \subseteq t_v(y) \)
- distributive: \( t_v(x \cup y) = t_v(x) \cup t_v(y) \)

Work List Iterative Algorithm

initialize \( \text{ReachingDefinitions}(n) \)

\text{worklist} \leftarrow \text{the set of all nodes}

\textbf{while} \text{ worklist} \neq \{ \}

- take \( n \) from \text{worklist}
- recompute \( \text{ReachingDefinitions}(n) \)
- if \( \text{ReachingDefinitions}(n) \) changed
- \text{worklist} \leftarrow \text{worklist} \cup \text{SUCC}(n)

\textbf{end}

\textbf{end}

initialization

\text{IN}(v) \leftarrow \{ \}

\text{OUT}(v) \leftarrow \text{GEN}(v)

computation

\text{IN}(v) = \quad \text{OUT}(v) = \quad
For each vertex, find the set of variable definitions that might reach that vertex.

\( \text{GEN}(v) \) variable \( v \) may be defined or assigned to

\( \text{KILL}(v) \) variable \( v \) is defined, overwriting other definitions

1. \textbf{read} \( N \)
2. \textbf{call} \( \text{check}(N) \)
3. \( i \leftarrow 1 \)
4. \textbf{while} \( i < N \) \textbf{do}
5. \quad \( a[i] \leftarrow a[i] + i \)
6. \quad \( i \leftarrow i + 1 \)
7. \textbf{end}
8. \textbf{print} \( a[N] \)

<table>
<thead>
<tr>
<th></th>
<th>GEN</th>
<th>KILL</th>
<th>PRED</th>
<th>SUCC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( N_1 )</td>
<td>( N )</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>( N_2 )</td>
<td></td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>( i_3 )</td>
<td>( i )</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>3,7</td>
<td>5,8</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>( a_5 )</td>
<td></td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>( i_6 )</td>
<td>( i )</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>6</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

Questions

- Does this always terminate?
- What answer does it compute?
- How fast (or slow) is it?
Correctness and Quality of Solution

Does it compute the answer we want?

Definition: For each basic block \( b \)

\[ MOP(b) = \bigcap_{q \in \text{Paths}(b)} T_q(\top) \]

- Paths that reach a block are reachable in the control flow graph, which may be conservative
- Perfect Solution = meet over real paths taken during program execution
- \( MOP \leq \) Perfect Solution
- In some sense, \( MOP \) is best feasible solution
- Not guaranteed to achieve \( MOP \) solution unless transfer functions distributive

Quality of Solution

Maximal Fixed Point (MFP)

- Any iterative data-flow problem that satisfies admissible function requirements when it converges to a solution and terminates, will have reached a Maximal Fixed Point solution
- \( MFP \) is unique, regardless of order of propagation
- If distributive, \( MFP = MOP \)
- Otherwise, \( MFP \leq MOP \)
- So, \( MFP \leq MOP \leq \) Perfect Solution

Termination

Why does the iterative data flow algorithm terminate?

Sketch of proof for reaching definitions:

1. each node is initialized to \( \{ \} \)
2. a definition has only one statement that generates it
3. \( \mathcal{F} \) is associative \( \Rightarrow \) \( \mathcal{F} \) is monotone \( \Rightarrow \) each \( x \in \text{reaching definitions} \) can be added once
4. \( N \times (E + 1) \) trips to take a definition to every node

Consequence of finite descending chain property

Question: How do we generalize this proof?
How fast can we make the iterative algorithm?

To avoid unnecessary work:
- Bound number of visits by visiting a node roughly after all its predecessors
  (reverse PostOrder for forward data-flow problem; conceptually, PostOrder for backward problem)
- Change to algorithm:
  
  ```
  changed ← false
  do
    foreach \( v \in V \) in rPostOrder do
      solve for \( b \)
      if \( \text{old} \neq \text{new} \)
        changed ← true
      end
  end
  while changed
  ```
- How does this improve performance?

Examples

PostOrder and Reverse PostOrder

Step 1: PostOrder

```latex
\textbf{proc} \text{main()} \equiv
\text{count} ← 1
Visit(\text{Entry})
\textbf{end}
```

```latex
\text{proc} \text{Visit}(v) \equiv
\text{mark} v \text{ as visited}
\textbf{foreach} \text{successor} \( s \) of \( v \) not yet visited
Visit(\( s \))
\textbf{end}
PostOrder(\( v \)) ← \text{count} + +
\textbf{end}
```

Step 2: rPostOrder

```
\textbf{foreach} \( v \in V \) do
  rPostOrder(\( v \)) ← | V | - PostOrder(\( v \))
\textbf{end}
```

Depth-first search \( \approx \) rPostOrder
Analysis of Data-flow Frameworks

Key things to look for in a data-flow framework

- the domain and its size
- size of a single fact
- forward or backward problem
- model of characteristic function

Representation

- Sets represented by *bit vector*
- **Size of each bit vector:**
  - **Available Expressions:** # distinct expressions in program
  - **Reaching Definitions:** # definitions in program
  - **Live Variable Analysis:** # variables in program

Complexity

- distinguish bit-vector steps from logical steps
- watch out for complex mappings (\textit{GEN} \rightarrow \textit{KILL})

Summary

- Iterative data-flow framework used to solve global data-flow problems
- Use semi-lattice to represent facts
- Analysis on semi-lattice with finite descending chains and monotone data-flow framework guarantees termination
- Monotonic data-flow framework guarantees \textit{MFP} solution reached
- Distributive to guarantee \textit{MOP} solution reached
- \textit{rPostOrder} (or \textit{PostOrder}) for "rapid" data-flow problems guarantees bound of $O(n(d + 2))$ complexity, where $d$ is maximum number of retreating edges on any acyclic path in the CFG (loop "interconnectiveness")