**CS 412/413**
Introduction to Compilers and Translators
Andrew Myers
Cornell University

Lecture 26: Dataflow analyses
3 April 00

---

**Need for dataflow analysis**
- Most optimizations require program analysis to determine safety
- This lecture: dataflow analysis
- Standard program analysis framework

---

**Dataflow analyses**
- **Live variable analysis** — register allocation, dead-code elimination
- **Reaching definitions**: what points in program does each variable definition reach? — copy, constant propagation
- **Available expressions**: which expressions computed earlier still have same value? — common sub-expression elimination

---

**IR for data-flow analysis**
- Tree IR: good for instruction selection, bad for data-flow analysis
- Can flatten tree representation into simple nodes (a,b,c temps, labels L)

---

**IR optimization**
- Canonical IR
- convert to flowgraph of quadruples
- convert to tree form
- analyze, optimize
- Abstract assembly
- Assembly code
- register allocation
- Instruction selection

---

**Converting to quadruples**
- Conversion is tree simplification that aggressively adds new temporaries

---
Converting back to tree

- Convert quadruples to simple trees
- Look for temporaries in statement sequence used and defined only once
- Move definition just before use
- Glue tree, eliminating temporary

\[
\begin{align*}
& t = c \times a \\
\quad \text{MOVE}(t, *\langle c, a \rangle) \\
&a = b + t \\
\quad \text{MOVE}(a, +\langle b, t \rangle)
\end{align*}
\]

- Requires dataflow analyses to do right (reaching definitions, available expressions)

Control flow graph

- Simplification: generate quadruples directly, reconstruct trees from quadruples later for instruction selection
- Quadruple sequence is control flow graph (flowgraph)
- Nodes in graph: quadruples (not assembly statements)
- Edges in graph: ways to transfer control between quadruples (including fall-through)
- For node \( n \), \( \text{use}[n] \) is variables used, \( \text{def}[n] \) is variables defined (assigned)

Def & Use

<table>
<thead>
<tr>
<th>( n )</th>
<th>( \text{def}[n] )</th>
<th>( \text{use}[n] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a = b ) OP ( c )</td>
<td>( a )</td>
<td>( b, c )</td>
</tr>
<tr>
<td>( a = [b] )</td>
<td>( a )</td>
<td>( b )</td>
</tr>
<tr>
<td>( [a] = b )</td>
<td>( a, b )</td>
<td></td>
</tr>
<tr>
<td>goto ( L ) if ( a ) goto ( L_1 ) else goto ( L_2 ) ( L: ) ( a = f(\ldots) )</td>
<td>( a )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( f(\ldots) )</td>
<td>( \ldots )</td>
<td></td>
</tr>
</tbody>
</table>

Live variable analysis

- Useful even for IR: dead code elimination
- Output: \( \text{in}[n] \) and \( \text{out}[n] \) associated with every node \( n \) in flowgraph
- Constraints:
  \[
  \text{in}[n] \supseteq \text{use}[n] \\
  \text{in}[n] \cup \text{def}[n] \supseteq \text{out}[n] \\
  \text{out}[n] \supseteq \text{in}[n'] \text{ for all successors } n' \text{ of } n
  \]
- Dataflow equations:
  \[
  \text{in}[n] = \text{use}[n] \cup (\text{out}[n] - \text{def}[n]) \\
  \text{out}[n] = \bigcup_{n'} \text{in}[n']
  \]

Reaching definitions analysis

- Question: what uses in program does a given variable definition reach?
- Used for constant propagation, copy propagation
  - if only one definition reaches a particular use, can replace use by definition
  - copy propagation requires that copied value still has same value – use available expressions
- Input: flowgraph
- Output: \( \text{in}[n] \), \( \text{out}[n] \) is set of nodes defining some variable such that defn may reach beginning, end of \( n \)

Reaching definitions

\[
\begin{align*}
& W : b = a + 2 \quad \text{out: } W \\
& X : c = b \times b \quad \text{in: } W, \text{out: } X, W \\
& Y : b = c + 1 \quad \text{in: } X, W, \text{out: } X, Y \\
& Z : \text{return } b \times a \quad \text{in: } X, Y
\end{align*}
\]
Gen, kill

- Define: \( \text{defs}(x) \) is set of nodes defining var \( x \)
- Define: \( \text{gen}[n], \text{kill}[n] \)

<table>
<thead>
<tr>
<th>( n )</th>
<th>( \text{gen}[n] )</th>
<th>( \text{kill}[n] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a = b ) O ( c )</td>
<td>( { n } )</td>
<td>( { } )</td>
</tr>
<tr>
<td>( a = ) [( b )]</td>
<td>( { n } )</td>
<td>( { } )</td>
</tr>
<tr>
<td>( [a] = b ) goto ( L )</td>
<td>( { } )</td>
<td>( { } )</td>
</tr>
<tr>
<td>if a goto ( L_1 ) else goto ( L_2 ) ( L: )</td>
<td>( { } )</td>
<td>( { } )</td>
</tr>
<tr>
<td>( a = f(\ldots) ) ( f(\ldots) )</td>
<td>( { n } )</td>
<td>( { } )</td>
</tr>
</tbody>
</table>

Solution Constraints

- \( \text{out}[n] \supseteq \text{gen}[n] \)
  - “A definition made by \( n \) at least reaches \( n \’ s \) output”
- \( \text{in}[n] \supseteq \text{out}[n] \) (if \( n \’ s \) is succ. of \( n \)
  - “definitions reach node \( n \’ s \) if they exit any predecessor \( n \”
- \( \text{out}[n] \cup \text{kill}[n] = \text{in}[n] \)
  - “A definition that reaches the input either reaches the output or is killed”

Data-flow equations

- \( \text{in}[n] = \bigcup_{n \in \text{prev}(n)} \text{out}[n] \)
- \( \text{out}[n] = \text{gen}[n] \cup (\text{in}[n] - \text{kill}[n]) \)
- Algorithm: init \( \text{in}[n], \text{out}[n] \) with empty sets, apply equations as assignments until no progress (usual representation: bit vector)
- Eventually all equations satisfied
- Will terminate because \( \text{in}[n], \text{out}[n] \) can only grow, can be no larger than set of all defs
- Finds minimal solution to constraint eqns: accurate

Def-use chains

- Reaching definitions tells which nodes a def can reach
- If node uses same variable, definition affects node (conservatively)
- Def-use (du-) chain: def node + all nodes with affected uses
- Use-def (ud-) chain: use node + all nodes with def that might affect use

du-, ud-chains

\[
\begin{align*}
\text{DU} \\
\text{b}: (W \rightarrow X), (Y \rightarrow Z) \\
\text{c}: (X \rightarrow Y, Z)
\end{align*}
\]

\[
\begin{align*}
\text{UD} \\
\text{b}: (X \rightarrow W), (Z \rightarrow Y) \\
\text{c}: (Y \rightarrow X, Z \rightarrow X)
\end{align*}
\]

Webs

- du-chain, ud-chain intersect if share some use or definition
- web: maximal set of intersecting du, ud-chains
  - disjoint set union algorithm with path compression: nearly linear
- Same variable may comprise multiple non-interacting webs: permits more optimization
Webs
• Register allocation by webs avoids false conflicts
  
  ```
  int i;
  for (i = 0; i<n; i++) { ... }
  ...
  for (i = 0; i<n; i++) { ... }
  ```

  no use/def pairs!

• Two different webs: can allocate `i` to two different registers

Register allocation
1. use reaching definitions to compute all related uses and defs
2. compute distinct webs, rename all temporaries to web names
3. run live variable analysis
4. temporaries conflict if one is live when another is def'd (or both live on input)
5. Run graph coloring algorithm of previous lecture to allocate registers

Forward vs. Backward
• Liveness: backward analysis
  ```
  in[n] = use[n] \cup (out[n] – def[n])
  ```

  ```
  out[n] = \bigcup_{n' \in \text{succ}[n]} in[n']
  ```

• Reaching definitions: forward analysis
  ```
  out[n] = gen[n] \cup (in[n] – kill[n])
  ```

  ```
  in[n'] = \bigcup_{n \in \text{prev}[n']} out[n]
  ```

Dataflow analysis
• Most dataflow analyses characterized simply by
  –forward vs. backward analysis
  –`gen[n]`
  –`kill[n]`

  –Use of intersection vs. union when combining data from several nodes
    ```
    out[n] = gen[n] \cup (in[n] – kill[n])
    ```

    ```
    in[n'] = \bigcap_{n \in \text{prev}[n']} out[n]
    ```

Available expressions
• Idea: want to perform common subexpression elimination

  ```
  a = x+1
  ... a = x+1
  b = x+1
  ```

  ```
  b = a
  ```

  Transformation is safe if original `x+1` is available

Dataflow values
• Let `in[n]`, `out[n]` be sets of nodes whose computed expression is available at `n`

  ```
  n \quad \text{gen}[n] \quad \text{kill}[n]
  a=b \quad \{\} – \text{kill}[n] \quad \text{uses}(a)
  a=[b] \quad \{\} – \text{kill}[n] \quad \text{uses}(a)
  [a]=b \quad \{\} \quad \text{uses}(\{x\}) \text{ (for all } x \text{ that may be equal to } a\}
  a=f(b,\ldots,b_n) \quad \{\} \quad \text{uses}(\{x\}) \text{ (for all } x\}
  ```

other \quad \{\}
**Constraints**

- \( \text{out}[n] \supseteq \text{gen}[n] \)
  - “An expression made available by \( n \) at least reaches \( n \)’s output”
- \( \text{in}[n'] \subseteq \text{out}[n] \) (if \( n' \) is succ. of \( n \))
  - “An expression is available at \( n' \) only if it is available at every predecessor \( n \)”
- \( \text{out}[n] \cup \text{kill}[n] \supseteq \text{in}[n] \)
  - “An expression available on input is either available on output or killed”

**Dataflow equations**

- \( \text{out}[n] \supseteq \text{gen}[n] \)
- \( \text{in}[n'] \subseteq \text{out}[n] \) (if \( n' \) is succ. of \( n \))
- \( \text{out}[n] \cup \text{kill}[n] \supseteq \text{in}[n] \)

Equations for iterative solution:

- \( \text{out}[n] = \text{gen}[n] \cup (\text{in}[n] - \text{kill}[n]) \)
- \( \text{in}[n'] = \bigcap_{n \in \text{pred}[n']} \text{out}[n] \)

\( \cap = \bigcap \) Starting condition:

- \( \text{in}[n] \) is set of all nodes
- \( \text{in}[\text{start}] = \{\} \)

**Summary**

- Tree IR makes dataflow more difficult
- Saw reaching definitions, available expressions analyses
- How to use reaching definitions for better register allocations via webs
- *Next time:* a theory to explain why iterative solving works