CSC410
Data Flow Analyses

AZADEH FARZAN
FALL 2017
Dataflow Analyses

Goal and scope:
- Calculate over-approximate information about the behaviours of the program.
- Do it efficiently!

Applications:
- Widely used in optimizing compilers
- Verification
- Testing
Let’s start with an example...
Dead Code Elimination

Can part of this program be removed without affecting program behaviour?

```
x := 2
y := 5
x := 1
z := 0
if (x <= 0) {
    z := x + 2
} else {
    z := y * y
}
x := z
```

Based on the principle that these assignments to \texttt{x} and \texttt{z} can never be read by any other statement.
Live Variable Analysis
Live Variable Analysis

Intuition: A variable $v$ is live at a program point if its value may be read later in some execution of the program. Otherwise, it is dead.

An assignment $x := exp$ can be safely removed from the program (i.e is dead code) if $x$ is dead (at the program point of this statement).
How do we calculate this information automatically and efficiently?
A Control Flow Graph (CFG) is a graphical representation of the executions of a program. Formally, a CFG is a directed graph \( \langle V, E \rangle \), where:

- \( V \) is a set of program labels (AKA locations, points)
- \( E \) is the control flow successor relation: if \( v \) follows \( u \) in some execution, then \( u \rightarrow v \in E \)

Every execution corresponds to a path in the CFG:

- the converse does not hold!
- A CFG represents an overapproximation of the executions of a program
Intuition: A variable $v$ is live at a program point if its value may be read later in some execution of the program. Otherwise, it is dead.

Formal definition: A variable $\text{var}$ is live at a label $l$ if there exists a CFG path from $l$ to a label $l'$ such that $l'$ reads $\text{var}$, and no label along that path writes to $\text{var}$. Otherwise it is dead.
Example

At $x := 1$:

- $x$ is live
- $y$ is live
- $z$ is dead
Let’s start automating this in the simplest context ...
Live Variable Analysis on Paths

Define for each label \( l \):

- \( \text{read}(l) \): the set of variables \( l \) reads from.
- \( \text{write}(l) \): the set of variables \( l \) writes to.

For a path \( l_1 \ldots l_n \):

\[
\text{live}(l_n) = \emptyset \\
\text{live}(l_{i-1}) = \text{live}(l_i) \setminus \text{write}(l_i) \cup \text{read}(l_i)
\]
Live Variable Analysis on Paths

\[
\begin{align*}
live(l_n) &= \emptyset \\
live(l_{i-1}) &= live(l_i) \setminus write(l_i) \cup read(l_i)
\end{align*}
\]

\[
\begin{align*}
live_{exit}(l_n) &= \emptyset \\
live_{entry}(l) &= live_{exit}(l) \setminus write(l) \cup read(l) \\
live_{exit}(l_{i-1}) &= live_{entry}(l_i)
\end{align*}
\]
What if I have branches along the way?
Live Variable Analysis on DAGs

\[ x \leq 0 \]

\[ z := x + 2 \]

\[ z := y \]

\[ y \times y \]

\[ x := z \]

Live on entry: \( \{ x,y \} \cup \{ x \} \)

Live on exit: \( \{ x,y \} \)

Live on entry: \( \{ x \} \)

Live on exit: \( \{ z \} \)

Live on entry: \( \{ y \} \)

Live on exit: \( \{ z \} \)

Live on entry: \( \{ x,y \} \)

Live on exit: \( \emptyset \)
Live Variable Analysis on DAGs

\[
\begin{align*}
\text{live}_{\text{entry}}(\ell) &= \text{live}_{\text{exit}}(\ell) \setminus \text{write}(\ell) \cup \text{read}(\ell) \\
\text{live}_{\text{exit}}(\ell) &= \bigcup_{\ell \rightarrow \ell' \in E} \text{live}_{\text{entry}}(\ell')
\end{align*}
\]
What if there are cycles in the graph?
Consider the equations below as a system of constraints. A solution is any assignment of live sets that satisfies the equations below:

\[
\begin{align*}
LV_{entry}(\ell) &= LV_{exit}(\ell) \setminus \text{write}(\ell) \cup \text{read}(\ell) \\
LV_{exit}(\ell) &= \bigcup_{\ell \rightarrow \ell' \in E} LV_{entry}(\ell')
\end{align*}
\]

Our desired live variables information certainly is a solution to this constraint system.
Solution not necessarily unique

\[ \begin{align*}
LV_{entry}(\ell) &= LV_{exit}(\ell) \setminus write(\ell) \cup read(\ell) \\
LV_{exit}(\ell) &= \bigcup_{\ell \rightarrow \ell' \in E} LV_{entry}(\ell')
\end{align*} \]

\[
\begin{array}{|c|c|c|}
\hline
\text{Label} & LV_{exit} & LV_{entry} \\
\hline
i := 0 & \{i,k\} & \{k\} \\
\hline
j := 1 & \{i,j,k\} & \{i,k\} \\
\hline
i < 10 & \{i,j,k\} & \{i,j,k\} \\
\hline
i := i + j & \{i,k\} & \{i,j,k\} \\
\hline
i := 0 & \emptyset & \emptyset \\
\hline
\end{array}
\]
Key Observation

Any solution $\text{Live}_{\text{entry}}$, $\text{Live}_{\text{exit}}$ to this system of equations overapproximates the true set of live variables.

$x$ is live at $l \Rightarrow x \in \text{LV}_{\text{exit}}(l)$

The desired solution is the least solution!
Can we compute the least solution?
The Worklist Algorithm

Algorithm 1 Worklist method

\[
\begin{align*}
\text{worklist} & \leftarrow \text{empty} \\
\text{for} \text{ each } l \in V \text{ do} & \\
& \quad \text{LV}_{\text{exit}}(l) \leftarrow \emptyset \\
& \quad \text{LV}_{\text{entry}}(l) \leftarrow \text{LV}_{\text{exit}}(l) \setminus \text{write}(l) \cup \text{read}(l) \\
& \quad \text{add } l \text{ to worklist} \\
\text{end for} \\
\text{while} \text{ worklist not empty do} & \\
& \quad \text{pick } l \text{ from worklist} \\
& \quad \text{old} \leftarrow \text{LV}_{\text{entry}}(l) \\
& \quad \text{LV}_{\text{exit}}(l) \leftarrow \bigcup_{l \rightarrow l' \in E} \text{LV}_{\text{entry}}(l') \\
& \quad \text{LV}_{\text{entry}}(l) \leftarrow \text{LV}_{\text{exit}}(l) \setminus \text{write}(l) \cup \text{read}(l) \\
& \quad \text{if } \text{old} \neq \text{LV}_{\text{entry}}(l) \text{ then} \\
& \quad \quad \text{Add predecessors of } l \text{ to worklist} \\
& \quad \text{end if} \\
\text{end while}
\end{align*}
\]
Worklist Algorithm Example

\[ LV_{entry}(\ell) = LV_{exit}(\ell) \setminus \text{write}(\ell) \cup \text{read}(\ell) \]
\[ LV_{exit}(\ell) = \bigcup_{\ell \rightarrow \ell' \in E} LV_{entry}(\ell') \]

<table>
<thead>
<tr>
<th>Label</th>
<th>( LV_{exit} )</th>
<th>( LV_{entry} )</th>
<th>In WL</th>
</tr>
</thead>
<tbody>
<tr>
<td>( i := 0 )</td>
<td>{i}</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>( j := 1 )</td>
<td>{i, j}</td>
<td>{i}</td>
<td>{i}</td>
</tr>
<tr>
<td>( i &lt; 10 )</td>
<td>{i, j}</td>
<td>{i, j}</td>
<td>{i, j}</td>
</tr>
<tr>
<td>( i := i + j )</td>
<td>{i}</td>
<td>{i, j}</td>
<td>{i, j}</td>
</tr>
<tr>
<td>( j := i )</td>
<td>{i, j}</td>
<td>{i}</td>
<td>{i}</td>
</tr>
<tr>
<td>( j := i )</td>
<td>{i, j}</td>
<td>{i}</td>
<td>{i}</td>
</tr>
<tr>
<td>( j := 0 )</td>
<td>( \emptyset )</td>
<td>{i, j}</td>
<td>( \emptyset )</td>
</tr>
</tbody>
</table>
Correctness
Worklist Algorithm Correctness

Why does the worklist algorithm always terminate?

- $L V_{\text{entry}}(l)/L V_{\text{exit}}(l)$: always increase.
- Chains in $2^{\text{vars}}$ are finite.
- Either the sets increase, or the worklist decreases.

Why does the worklist algorithm compute the least solution?

- Loop invariant for the algorithm:
  
  \[
  \begin{align*}
  L V_{\text{exit}}(l) & \subseteq live_{\text{exit}}(l) \\
  L V_{\text{entry}}(l) & \subseteq live_{\text{entry}}(l)
  \end{align*}
  \]