Lecture 19: Live Variable Analysis
Problem

- Abstract assembly contains arbitrarily many registers $t_i$
- Want to replace all such nodes with register nodes
- Local variables allocated to TEMP’s too
- Only 9–15 usable registers: need to allocate multiple $t_i$ to each register
- For each statement, need to know which variables are live to reuse registers
Using scope

• Observation: temporaries, variables have bounded scope in program
• Simple idea: use information about program scope to decide which variables are live
• Problem: overestimates liveness

```c
{ int b = a + 2;
  int c = b*b;
  int d = c + 1;
  return d; }
```

b is live

͛ c is live, b is not

what is live here?
Live variable analysis

• Goal: for each statement, identify which temporaries are live

• Analysis will be conservative (may over-estimate liveness, will never under-estimate)

But more precise than simple scope analysis (will estimate fewer live temporaries)
Control Flow Graph

- Canonical IR forms control flow graph (CFG): statements are nodes; jumps, fall-throughs are edges

![Diagram of Control Flow Graph]

- fall-through edges
- in-edges
- out-edges
Liveness

- Liveness is associated with *edges* of control flow graph, not nodes (statements)

- Same register can be used for different temporaries manipulated by one stmt
Example

\[
a = b + 1
\]

\[
\text{MOVE(TEMP(ta), TEMP(tb) + 1)}
\]

\[
\text{mov tb, ta}
\]

\[
\text{add $1, ta}
\]

Register allocation: ta ⇒ rax, tb ⇒ rax

\[
\text{mov rax, rax}
\]

\[
\text{add $1, rax}
\]

Live: tb

Live: ta (maybe)
Use/Def

• Every statement uses some set of variables (reads from them) and defines some set of variables (writes to them)
  • For statement $s$ define:
    – $use[s]$: set of variables used by $s$
    – $def [s]$: set of variables defined by $s$
  • Example:

$$a = b + c \quad \begin{align*}
use &= b, c \\
def &= a
\end{align*}$$

$$a = a + 1 \quad \begin{align*}
use &= a \\
def &= a
\end{align*}$$
Liveness

Variable \( v \) is live on edge \( e \) if:

There is

– a node \( n \) in the CFG that uses it \textit{and}

– a directed path from \( e \) to \( n \) passing through \textit{no def}

How to compute efficiently?
How to use?
Simple algorithm: Backtracing

“variable \( v \) is live on edge \( e \) if there is a node \( n \) in CFG that uses it and a directed path from \( e \) to \( n \) passing through no def for \( v \)”

(Slow) algorithm: Try all paths from each use of a variable, tracing backward in the control flow graph until a def node or previously visited node is reached. Mark variable live on each edge traversed.
Dataflow Analysis

• Idea: compute liveness for all variables simultaneously
  • Approach: define equations that must be satisfied by any liveness determination
  • Solve equations by iteratively converging on solution
  • Instance of general technique for computing program properties: dataflow analysis
Dataflow values

use\([n]\) : set of variables used by \(n\)
def\([n]\) : set of variables defined by \(n\)
in\([n]\) : variables live on entry to \(n\)
out\([n]\) : variables live on exit from \(n\)

Clearly: \(\text{in}[n] \supseteq \text{use}[n]\)

What other constraints are there?
Dataflow constraints

\[ \text{in}[n] \supseteq \text{use}[n] \]

- A variable must be live on entry to \( n \) if it is used by the statement itself

\[ \text{in}[n] \supseteq \text{out}[n] \quad \text{– def } [n] \]

- If a variable is live on output and the statement does not define it, it must be live on input too

\[ \text{out}[n] \supseteq \text{in}[n'] \quad \text{if } \quad n' \in \text{succ } [n] \]

- if live on input to \( n' \), must be live on output from \( n \)
Iterative dataflow analysis

• Initial assignment to $in[n]$, $out[n]$ is empty set $\emptyset$ : will not satisfy constraints

$$in[n] \supseteq use[n]$$

$$in[n] \supseteq out[n] - def[n]$$

$$out[n] \supseteq in[n'] \text{ if } n' \in succ[n]$$

• Idea: iteratively re-compute $in[n]$, $out[n]$ when forced to by constraints. Live variable sets will increase monotonically.

• Dataflow equations:

$$in'[n] = use[n] \cup (out[n] - def[n])$$

$$out'[n] = \bigcup_{n' \in succ[n]} in[n']$$
Complete algorithm

for all $n$, $\text{in}[n] = \text{out}[n] = \emptyset$
repeat until no change
    for all $n$
        $\text{out}[n] = \bigcup_{n' \in \text{succ}[n]} \text{in}[n']$
        $\text{in}[n] = \text{use}[n] \cup (\text{out}[n] - \text{def}[n])$
    end
end

• Finds *fixed point* of in, out equations
• But: does extra work recomputing in, out values when no change can happen
For simplicity: pseudo-code

```
e=1
if x>0
  z=e*e
  y=e*x
  if x&1
    e=y
  return x
```

- **def:** e
- **use:** x
- **def:** z
- **use:** e, x
- **def:** y
- **use:** e
- **use:** x
- **def:** e
- **use:** y
- **def:** e
- **use:** y
Example

```python
e = 1
if x > 0
    z = e * e
    y = e * x
if x & 1
    e = y
return x
```

def: e
use: x
use: x
use: e

def: z
use: e, x

def: y
use: e, x

def: e
use: z

use: z

use: y

use: x

use: e

all equations satisfied

Faster algorithm

• Information only propagates between nodes because of this equation:

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

• Node is updated from its successors
  – If successors haven’t changed, no need to apply equation for node
  – Should start with nodes at “end” and work backward
Worklist algorithm

- Idea: keep track of nodes that might need to be updated in *worklist*: FIFO queue

for all n, $\text{in}[n] = \text{out}[n] = \emptyset$

$w = \{ \text{set of all nodes} \}$

repeat until $w$ empty

\[
\begin{align*}
    & n = w.\text{pop}( ) \\
    & \text{out}[n] = \bigcup_{n' \in \text{succ}[n]} \text{in}[n'] \\
    & \text{in}[n] = \text{use}[n] \cup (\text{out}[n] - \text{def}[n]) \\
    & \text{if change to } \text{in}[n], \\
    & \quad \text{for all predecessors } m \text{ of } n, w.\text{push}(m)
\end{align*}
\]

end
Running time

• out[n] can change at most V times where V is the number of variables.

• How many times is a node pushed onto the worklist?
  – once at beginning
  – at most once for each time successor nodes are updated; at most 2 successors

• Total pushes per node: at most 2V+1.

⇒ O(NV) total node updates