CS 4120
Introduction to Compilers
Ross Tate
Cornell University

Lecture 26: Live-Variable Analysis

Problem
- Abstract assembly contains arbitrarily many registers \( t \)
- Want to replace all such nodes with register nodes for \( e[a-d]x, e[bdj], (ebp) \)
- Local variables allocated to TEMP’S too
- Only 6-7 usable registers: need to allocate multiple \( t \) 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

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

Liveness
- Liveness is associated with edges of control flow graph, not nodes (statements)
- Same register can be used for different temporaries manipulated by one statement
Example

\[ a = b + 1 \]

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

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

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

Register allocation: \( ta \rightarrow \text{eax}, \tb \rightarrow \text{eax} \)

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

\[ \text{add eax}, 1 \]

Live: \( tb \)

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

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

Live: \( ta \)

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:
  - \( \text{use}[s] \): set of variables used by \( s \)
  - \( \text{def}[s] \): set of variables defined by \( s \)
- Example:
  - \( a = b + c \)
    - \( \text{use} = b, c \)
    - \( \text{def} = a \)
  - \( a = a + 1 \)
    - \( \text{use} = a \)
    - \( \text{def} = a \)

Liveness

- Variable \( v \) is live on edge \( e \) if there is
  - a node \( n \) in the CFG that uses it and
  - a directed path from \( e \) to \( n \) passing through no \( \text{def} \)
- How to compute efficiently?
- How to use?

Simple algorithm: Backtracing

- "variable \( v \) is live on edge \( e \) if there is a node \( n \) in the CFG that uses it and a directed path from \( e \) to \( n \) passing through no \( \text{def} \)"
- (Slow) algorithm: Try all paths "from" each use of a variable, tracing backward in the CFG until a \( \text{def} \) 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 formulae that must be satisfied by any liveness determination
- Solve formulae by iteratively converging on solution
- Instance of general technique for computing program properties: data-flow analysis

Data-flow values

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

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

What other constraints are there?
**Data-flow constraints**

- $in[n] \supseteq use[n]$
  - A variable must be live on entry to $n$ if it is used by the statement itself
- $in[n] \supseteq out[n] \setminus def[n]$
  - If a variable is live on output and the statement does not define it, it must be live on input too
- $out[n] \supseteq in[n']$ if $n' \in succ[n]$
  - if live on input to $n'$, must be live on output from $n$

**Iterative data-flow analysis**

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

  $$in[n] \supseteq use[n]$$
  $$out[n] \supseteq out[n'] \setminus def[n']$$

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

  - Dataflow equations:
    $$in'[n] = use[n] \cup (out[n] \setminus def[n])$$
    $$out'[n] = \bigcup_{n' \in succ[n]} in[n']$$

**Complete algorithm**

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

- Finds fixed point of in/out equations
- Problem: does extra work recomputing in/out values when no change can happen

**Example**

$e=1$
if $x>0$
$z=e*e$
y=$e*x$
e=z$
if $x&1$
e=$y$
return $x$
def: $e$
use: $x$
use: $x$
use: $e$
def: $z$
use: $e$, $x$
def: $y$
use: $z$
def: $e$
use: $x$
def: $e$
use: $y$
def: $e$
all equations satisfied

$2$: $in=[x]$
$3$: $in=[e]$
$4$: $in=[x]$
$5$: $in=[e,x]$
$6$: $in=[x]$
$7$: $out=[x], in=[x,z]$
$8$: $out=[x], in=[y,x]$  
$1$: $out=[x], in=[x]$  
$5$: $out=[x], in=[e,x]$  
$6$: $out=[x], in=[e,x]$  
$7$: $out=[e,x], in=[e,x]$  
$8$: $out=[e,x], in=[e,x]$  
$1$: $out=[e,x], in=[e,x]$  
$5$: $out=[e,x], in=[e,x]$  
$6$: $out=[e,x], in=[e,x]$  
$7$: $out=[e,x], in=[e,x]$  
$8$: $out=[e,x], in=[e,x]$  
all equations satisfied

**Faster algorithm**

- Information only propagates between nodes because of this equation:
  $$out[n] = \bigcup_{n' \in succ[n]} in[n']$$

- 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 $n$, $in[n] = out[n] = \emptyset$
  w = [ set of all nodes ]
  repeat until w empty
    n = w.pop()
    $$out[n] = \bigcup_{n' \in succ[n]} in[n']$$
    $$in[n] = use[n] \cup (out[n] \setminus def[n])$$
    if change to $in[n]$
      for all predecessors m of n, w.add(m)
  end