CS 412
Introduction to Compilers

Andrew Myers
Cornell University

Lecture 25: Live Variable Analysis
2 Apr 01

Administration

- Programming Assignment 4 due this Friday

Outline

- Register allocation problem
- Liveness
- Liveness constraints
- Solving dataflow equations
- Interference graphs

Problem

- Abstract assembly contains arbitrarily many registers t_i
- Want to replace all such nodes with register nodes for `e[a-d]x`, `e[sd]i`, `(ebp)`
- Local variables allocated to `TEMP`'s too
- Only 6-7 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

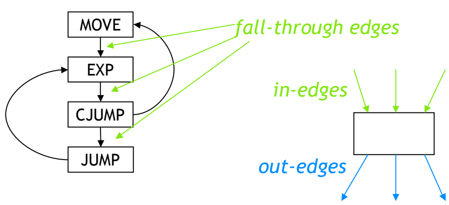
```
{  int b = a + 2;   <-- b is live
   int c = b*b;     <-- c is live, b is not
   int d = c + 1;
   return d; }      <-- what is live here?
```


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)

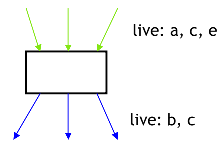
Control Flow Graph

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



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

⇩

MOVE(TEMP(ta), TEMP(tb) + 1)

⇩

mov ta, tb
add ta, 1

Live: tb
mov ta, tb
add ta,1
Live: ta (maybe)

Register allocation: ta ⇒ eax, tb ⇒ eax
~~mov eax, eax~~
add eax, 1

Use/Def

- Every statement *uses* some set of variables (read 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	*use* = b,c	*def* = a
a = a + 1	*use* = a	*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 *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*"

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

What other constraints are there?

Dataflow constraints

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

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

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

- 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 dataflow analysis

- Initial assignment to *in*[*n*], *out*[*n*] is empty set Ø : will not satisfy constraints

$$in[n] \supseteq use[n]$$
$$in[n] \supseteq out[n] - def[n]$$
$$out[n] \supseteq in[n'] \quad \text{if} \quad n' \in succ[n]$$

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

$$in'[n] = use[n] \cup (out[n] - def[n])$$
$$out'[n] = \cup_{n' \in succ[n]} in[n']$$

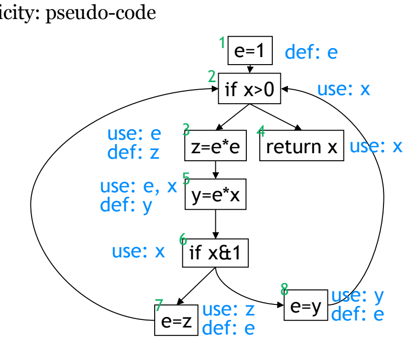
Complete algorithm

```
for all n, in[n] = out[n] = Ø
repeat until no change
    for all n
        out[n] = ∪n' ∈ succ[n] in[n']
        in[n] = use[n] ∪ (out[n] - def[n])
    end
end
```

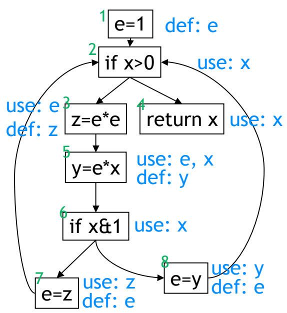
- Finds *fixed point* of in, out equations
- Problem: does extra work recomputing in, out values when no change can happen

Example

- For simplicity: pseudo-code



Example



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={x,y}
1: out={x}, in={x}
2: out={e,x}, in={e,x}
3: out={e,x}, in={e,x}
5: out={x}, in={e,x}
6: out={x,y,z}, in={x,y,z}
7: out={e,x}, in={x,z}
8: out={e,x}, in={x,y}
1: out={e,x}, in={x}
5: out={x,y,z}, in={e,x,z}
3: out={e,x,z}, in={e,x}
all equations satisfied

Faster algorithm

- Information only propagates between nodes because of this equation:

  out[n] = $\bigcup_{n' \in \text{succ}[n]}$ 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] = Ø
w = { set of all nodes }
repeat until w empty
    n = w.pop( )
    out[n] = ∪ n' ∈ succ [n] in[n']
    in[n] = use[n] ∪ (out[n] — def [n])
    if change to in[n],
        for all predecessors m of n, w.push(m)
end
```

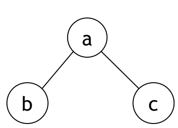

Register allocation

- For every node *n* in CFG now have *out*[*n*] : which variables (temporaries) are live on exit from node.
  - Also consider *in*[*start*]
- If two variables are in same live set, can't be allocated to the same register –they *interfere* with each other
- How do we assign registers to variables?

Interference graph

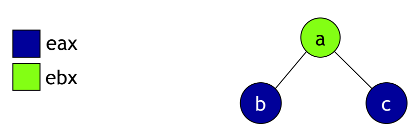
- Undirected graph of variables
- Construct graph with one node for every variable
- Add edge between every two variables that interfere with each other

```
b = a + 2;      a
c = b*b;        a,b
b = c + 1;      a,c
return b*a;     a,b
```



Graph coloring

- Problem: assign a register to every node in graph, but connected nodes cannot be given the same register
- *Graph coloring* problem : can we color the interference graph using 6-7 colors?



Summary

- Live variable analysis tells us which variables we need to have values for at various points in program
- Liveness can be computed by backtracing or by dataflow analysis
- Dataflow analysis finds solution iteratively by converging on solution
- Register allocation is coloring of interference graph