**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 \( e[a-d], e[sd], (ebp) \)
- 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

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

---

**Live variable analysis**

- Goal: for each statement, identify which temporaries are live
- Analysis will be **conservative** (may overestimate liveness, will never underestimate)
- 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

Example

\[ a = b + 1 \]

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

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

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

Live: \( ta \) (maybe)

Register allocation: \( ta \Rightarrow eax \), \( tb \Rightarrow eax \)

Liveness

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

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

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 CFG that uses it and a directed path from \( e \) to \( n \) passing through no \( \text{def} \)"

Algorithm: Try all paths from each use of a variable, tracing backward in the control flow graph 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 *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] ⊇ use[n]`

What other constraints are there?

Dataflow constraints

- `in[n] ⊇ use[n]`
  - A variable must be live on entry to `n` if it is used by the statement itself
- `in[n] ⊇ 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] ⊇ in[n']` if `n' ∈ 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 ∅**: will not satisfy constraints

  - `in[n] ⊇ use[n]`
  - `in[n] ⊇ out[n] – def[n]`
  - `out[n] ⊇ in[n']` if `n' ∈ succ[n]`

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

  - **Dataflow equations**:
    
    \[
    \begin{align*}
    in' & = use[n] \cup (out[n] – def[n]) \\
    out' & = \bigcup_{n' \in succ[n]} in[n']
    \end{align*}
    \]

Complete algorithm

- **For all `n`, `in[n] = out[n] = ∅`**: repeat until no change
  - **For all `n`**
    
    \[
    \begin{align*}
    out[n] & = \bigcup_{n' \in succ[n]} in[n'] \\
    in[n] & = use[n] \cup (out[n] – def[n])
    \end{align*}
    \]
  - **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**

```
e = 1
if x > 0
    z = e * e
    y = e * x
else
    e = y
use: e
def: e
```
**Example**

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

**Faster algorithm**

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

- 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, in[n] = out[n] = Ø
  
  w = { set of all nodes }

  repeat until w empty
    n = w.pop()
    out[n] = \bigcup_{n'} \text{succ}[n] \cap \text{in}[n']
    in[n] = use[n] \cup (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 \( \text{out}[n] \): which variables (temporaries) are live on exit from node.
  
  - Also consider \( \text{in}[\text{start}] \)

- 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;
c = b*b;
b = c + 1;
return b*a;
```

**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