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 ti
- Want to replace all such nodes with register nodes for e[a-d], e[s][d], (ebp)
- Local variables allocated to TEMP’s too
- Only 6-7 usable registers: need to allocate multiple ti 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: over-estimates liveness

```
{ 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 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

```
MOVE
EXP
I
JUMP
```

fall-through edges

```
MOVE
EXP
I
JUMP
```

incoming edges

```
MOVE
EXP
I
JUMP
```

outgoing edges

Liveness

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

```
 live: a, c, e
 live: b, c
```

- 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
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
```

```
a = b + c
use = b, c  def = a
```

```
a = a + 1
use = a  def = a
```

Liveness

- Definition of 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?

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.

Running time?
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 \(\emptyset\) : will not satisfy constraints

\(in[n] \supseteq use[n]\)
\(in[n] \supseteq out[n] - def[n]\)
\(out[n] \supseteq in[n']\) 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 \(n\), \(in[n] = out[n] = \emptyset\)
repeat until no change
for all \(n\)

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

\[
\begin{align*}
\text{e} &= 1 \\
\text{if } x &> 0 \\
\text{z} &= e \times e \\
y &= e \times x \\
\text{e} &= z \\
\text{if } x \& 1 \\
y &= e \\
\text{return } x
\end{align*}
\]

\begin{itemize}
\item def: e
\item use: x
\item use: x
\item use: y
\item def: e
\item use: x
\item use: y
\item def: e
\end{itemize}

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

\begin{itemize}
\item Idea: keep track of nodes that might need to be updated in \text{worklist} : FIFO queue
\item for all \text{n}, \text{in}[][\text{n}] = \text{out}[][\text{n}] = \emptyset
\item \text{w} = \{ \text{set of all nodes} \}
\item \text{repeat until } \text{w} \text{ empty}
\item \text{n} = \text{w}.\text{pop}()
\item \text{out}[][\text{n}] = \bigcup_{n' \in \text{succ}[\text{n}]} \text{in}[][n']
\item \text{in}[][\text{n}] = \text{use}[][\text{n}] \cup (\text{out}[][\text{n}] - \text{def}[\text{n}])
\item \text{if change to } \text{in}[][\text{n}]
\item \text{for all predecessors } m \text{ of } \text{n}, \text{w}.\text{push}(m)
\end{itemize}

Faster algorithm

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

- For every node \text{n} in CFG now have \text{out}[][\text{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?

Register allocation

- For every node \text{n} in CFG now have \text{out}[][\text{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

\[
\begin{align*}
b &= a + 2; \\
c &= b^2; \\
b &= c + 1; \\
\text{return } b^a;
\end{align*}
\]

Graph coloring

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

\[
\begin{align*}
\text{eax} \\
\text{ebx}
\end{align*}
\]
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